Scalable Solutions to Integral Equation and Finite Element Simulations

Tom Cwik, Daniel S. Katz,* and Jean Patterson

Jet Propulsion Laboratory
4800 Oak Grove Drive
California Institute of Technology
Pasadena, California, 91109
cwik@jpl.nasa.gov

*Cray Research Inc.

ABSTRACT

When developing numerical methods, or applying them to the simulation and design of engineering components, it inevitably becomes necessary to examine the scaling of the method with a problem's electrical size. The scaling results from the original mathematical development- -- for example, a dense system of equations in the solution of integral equations-as well as the specific numerical implementation. Scaling of the numerical implementation depends upon many factors--for example, direct or iterative methods for solution of the linear system--as well as the computer architecture used in the simulation. In this paper, scalability will be divided into two components; scalability of the numerical algorithm specifically on parallel computer systems, and algorithm or sequential scalability. The sequential implementation and scaling is initial presented with the parallel implementation following. This progression is meant to illustrate the differences in using current parallel platforms from sequential machines, and the resulting savings. Time to solution (wall clock time) along with problem size are the key parameters plotted or tabulated. Sequential and parallel scalability of time harmonic surface integral equation forms and the finite element solution to the partial differential equations are considered in detail.

1. INTRODUCTION

The application of advanced computer architecture and software to a broad range of electromagnetic problems has allowed more accurate simulations of electrically larger and more complex components and systems. Computational algorithms are used in the design and analysis of antenna components and arrays, waveguide components, semiconductor devices, and in the prediction of radar scattering from aircraft, sea surface or vegetation, and atmospheric particles, among many other applications. A scattering algorithm, for example, may **be** used in conjunction with measured data to accurately reconstruct geophysical data. When used in design, the goal of a numerical simulation is to limit the number of trial fabrication and measurement iterations needed. The algorithms are used over a wide range of frequencies, materials and shapes, and can be developed to be specific to a single geometry and application, or more generally applied to a class of problems. The algorithms are numerical solutions to a mathematical model developed in the time or frequency domain, and can be based on the integral equation or partial differential equation form of Maxwell's equations. A survey of the many forms of mathematical modeling and numerical solution can be found in [1, 2.]. The goal of this paper is to examine the scalability of certain numerical solutions both generally as a sequential algorithm, and then specifically as parallel algorithms using distributed memory computers. Parallel computers continue to evolve, surpassing traditional computer architectures in offering the largest memories and fastest computational rates to the user, and will continue to evolve for several more generations in the near future [3].

This paper overviews solutions to Maxwell's equations implicitly defined through systems of linear equations. Sequential and parallel scalability of time harmonic surface integral equation forms and the finite element solution to the partial differential equations are considered. More general scalability of other sequential algorithms can be found in [4]. initially in Section 2, a short review of the scalability y of parallel computers is presented. in Sections 3 and 4 respectively, specific implementations of the method of moments solution to integral equation modeling, and a finite element solution wilt be discussed. The scalability of sequential solutions and related

reduced memory methods will be considered followed by parallel scalability and computer performance for these algorithms. The size of problems capable of being examined with current computer architectures will be then be listed, with scalings to larger size problems also presented.

2. SCALABILITY OF PARALLEL COMPUTERS

Parallel algorithm scalability is examined differently from sequential algorithm scalability, or more precisely, scalability on shared memory (common address space) machines with no inter-processor communication overhead. Since a parallel computer is an ensemble of processors and memory linked by high performance communicant ion networks, calculations that were performed on a single processor must be broken into pieces and spread over all processors in USC. At intermediate points in the calculation, data needed in the next stage of the calculation must be communicated to other processors.

The central consideration in using a parallel computer is to decompose the discretized problem among processors so that the storage and computational load arc balanced, and the amount of communication between processors is minimal [5,6]. When this is not handled properly, efficiency is lower than 100%, where 100% is the machine performance when all processors arc performing independent calculations and no time is used for communication. If the problem is decomposed poorly, some processors will work while others stand idle, thereby lowering machine efficiency. Similarly, if calculations are load balanced but processors must wait to communicate data, the efficiency is lowered. Scalability and efficiency arc defined to quantify the parallel performance of a machine. Scalability, also termed speedup, is the ratio of time to complete calculations sequentially on a single processor (o that on P processors

$$S = T_{eq}/T(P) \text{ ``}$$ (1)

The efficiency is then the ratio of scalability to the number of processors

$$\varepsilon = S/P \ .$$ (2)

If an algorithm issues no communication calls, and there is no component of the calculation that is sequential and therefore redundantly repeated at each processor, the scalability is equal to the number of processors P and the efficiency is 100%. The scalability, as defined, must be further clarified if it is to be meaningful since the amount of storage, i.e., problem size, has not been included in the definition. Two regimes can be considered--fixed problem size and fixed grain size. The first, fixed problem size, refers to a problem that is small enough to fit into one or a few processors and is successively spread over a larger sized machine. The amount of data and calculation in each processor will decrease and the amount of communication will increase. The efficiency must therefore successively decrease, reaching a point where CPU time is communication bound. The second, fixed grain size problems, refers to a problem size that is scaled to fill all the memory of the machine in use. The amount of data and calculation in each processor will be constant, and in general, much greater than the required amount of communication. Efficiency will remain high as successively larger problems are solved. Fixed grain problems ideally exhibit scalability that is a key motivator for parallel processing-successively larger problems can be mapped onto successively larger machines without a loss of efficiency.

When developing numerical methods, or applying them to the simulation and design of engineering components, it inevitably becomes necessary to examine the scaling of the method with a problem's electrical size. The scaling results from the original mathematical development—for example, a dense system of equations in the solution of integral equations-—as well as the specific numerical implementation. Scaling of the numerical implementation depends upon many factors- for example, direct or iterative methods for solution of the linear system- as well as the computer architecture used in the simulation. in the rest of this paper, scalability will be divided into two components; first, algorithmic scalability, and second, scalability of the numerical algorithm specifically on parallel computer systems. Algorithmic scalabilty refers to the amount of computer memory and time needed to complete an accurate solution as a function of the electrical size of the problem. Scalability on a parallel computer system refers to the ability of an algorithm

4

to achieve performance proportional to the number of processors being used as outlined above. The sequential implementation and scaling is initial presented with tile parallel implementation following. This progression is meant to illustrate the differences in using current parallel platforms from sequential machines, and the resulting savings. Clearly, different mathematical formulations can lead to alternate numerical implementations and different scalings. The objective of any numerical method is to provide an accurate simulation of the measurable quantities in an amount of time that is useful for engineering design. With this objective, time to solution (wall clock time) along with problem size arc the key parameters plotted or tabulated.

3. INTEGRAL EQUATION FORMULATIONS

The method of moments is a traditional algorithm used for the solution of a surface integral equation [7,8]. This technique can be applied to impenetrable and homogenous objects, objects where an impedance boundary condition accurately models inhomogenous materials or coatings, and those inhomogenous objects that allow the use of Green's functions specific to the geometry. A dense system of equations results from discretizing the surface basis functions in a piecewise continuous set, with this system being solved in various ways. The components of method of moment solutions that affect the algorithmic scalability arc the matrix fill, and solution. A system of equations

$$AX = B \tag{3}$$

results from the method of moments, where A generally is an non-symmetric complex-valued square matrix, and B and X arc complex-valued rectangular matrices when multiple excitations and solution vectors arc present. The solution of (3) is most conveniently found by an LU factorization

$$A = LU \tag{4}$$

where L and U arc lower and upper triangular factors of A. The solution for X is computed by successive forward and back substitutions to solve the triangular systems

$$LY = B, \quad UX = -Y. \tag{5}$$

Because the system in (3) in not generally positive definite, rows of A arc permuted in the factorization, leading to a stable algorithm [9]. Table 1 is a listing of computer storage and time scalings for a range of problem sizes when using standard LU decomposition factorization, and forward and backward solution algorithms readily available [10]. The table is divided along columns into problem size, factorization and solution components. The first column fixes the memory size of the machine being used; the nu mber of unknowns and surface area modeled (assuming 200 unknowns/λ^2), based on this memory arc in the next columns. The factor and solve times are based on the performance of three classes of machine, current high-end workstation (O. 1 Gigaflops), current supercomputer (10 Gigaflops), and next generation computer (1000 Gigaflops). Similarly, the rows are divided into the top four which would typically correspond to current generation workstations, and the lower four that correspond to supercomputer class machines. Due to the nature of the dense matrix data structures [10] in the factorization algorithms, this component of the calculation can be highly efficient on general computers. A value of 80% efficiency of the peak machine performance is used for the time scali rigs. The backward and forward solution algorithms though operate sequentially on triangular matrix systems, resulting in reduced performance, and a 50% efficiency is used in these columns. 'I'his performance will increase when many excitations (right-hand sides) arc involved in the calculation, resulting in performance closer to peak.

Table I. Scaling of method of moments matrix factorization and solution algorithms.

PROBLEM SIZE			FACTOR			SOLVE (1 excitation)		
MEMORY (Mbytes)	N	AREA (λ^2)	0.1	10	1000	0.1	10	1000
			Gflops machine performance			Gflops machine performance		
128	2,800	14	12.2 mins	0.12 mins	0.001 mins	1.25 secs	10^{-2} secs	10^{-4} secs
256	4,000	20	35.6	0.36	0.004	2.55	2.5×10^{-2}	2.5×10^{-4}
512	5,600	28	97.6	0.98	0.010	5.01	5.0×10^{-2}	5.0×10^{-4}
1024	8,000	40	284.4	2.84	0.028	10.21	0.10	0.001
8192	22,627	113	107.3 hrs	1.07 hrs	0.017 hrs	1.36 mins	10^{-2} mins	10^{-4} mins
32,768	45,254	226	-	8.58	0.086	-	5.5×10^{-2}	5.5×10^{-4}
131,072	"90,509	452	-	68.65	0.687	-	0.22	0.002
524,288	181,019	905	-	-	5.492	-	-	0.009

The time for factorization, scaling as N^3, can limit the time of the calculation for large problems, even if the matrix can be assembled and stored. An alternative to direct factorization methods is the use of iterative solvers of the dense system [11,12]. These methods are even more useful if multiple right hand sides are present, since the iterative solvers exploit information in these vectors [11]. If the convergence rate can be controlled, and the number of iterations required (o complete a solution are limited, the solution time can be reduced as compared to the direct factorization methods.

The direct limitation of integral equation methods though, is the memory needed to store the dense matrix. Larger problems can only be simulated by circumventing this bottleneck. One approach to this problem is to use higher-creier parametric basis functions which reduce the number of unknowns needed to model sections of the surface [13]. However, it is clear that the growth in memory required for storage of the dense complex impedance matrix (N^2), cannot be overcome by only slightly reducing the size of N or by increasing the number of computer processors applied to the problem. Alternative methods are then desirable.

One such method uses special types of basis functions to produce a sparse impedance matrix [14], reducing the storage from the N^2 to αN where α is a constant independent of N. The resulting sparse system **can be** solved using **tile** methods described in latter sections of this paper. Other **classes** of methods for gene.l-atinp, and solving the impedance matrix while reducing storage

-/

arc summarized in [15], while another is the. fast multipole method [16, 17]. This approach decomposes the impedance matrix in a manner that also reduces the needed storage to αN. The fast multipole method has been parallelized in [18], where it is suggested that an Intel Paragon system with 512 nodes each containing 32 Mbytes of memory could solve a problem with 250,000 basis functions.

3.a Scability on Parallel Computers

To specifically examine parallel stability, the electric-field integral equation model developed in the PATCH code [19,20] is used. This code uses a triangularly faceted surface model of the object being modeled, and builds a complex dense matrix system (identical to that given in equation (3))

$$Z \, I = V \tag{6}$$

where

$$Z_{i,j} = \iint_{\partial S} T_i(\bar{r})d\bar{r} \iint_{\partial S} J_j(\bar{r}')G(\bar{r} - \bar{r}')d\bar{r}' \tag{7}$$

and i and j arc indices on the edges of the surface facets, G is the Green's function for an unbounded homogeneous space, $\bar{r}$ and $\bar{r}'$ are arbitrary source and observation points, and { T_i, T_j} arc current expansion and testing functions. The impedance matrix (Z) is factored by means of an LU factorization, and for each vector V a forward and backward substitution is performed to obtain 1, the unknown currents on the edges of the surface facets. It is then a simple matter to compute radar cross section or other field quantities by a forward integral. The elements of the PATCH code considered in the parallelization arc matrix fill, matrix factorization and solution of one or many right hand sides, and the calculation of field quantities. The computation cost of these three elements must be examined in relation to increasing problem size and increasing number of processors in order to understand the scalability of the I'Al'C} I code.

3.a.i Matrix Equation Fill

Since the impedance matrix is a complex dense matrix of size N, where N is the number of edges used in the faceted surface of the object being modeled, the matrix has N^2 elements, and filling this matrix scales as N^2. One method for reducing the amount of time spent in this operation when using a parallel computer is to spread the fixed number of elements to be computed over a large number of processors. Since the amount of computation is theoretically fixed (neglecting communication between processors), applying P processors to this task should provide a time reduction of P. However, the computations required by the PATCH code's basis functions involve calculations performed at the center of each patch that contribute to the matrix elements associated with the three current functions on the edges of that patch. The sequential PATCH code will loop over the surface patches and compute partial integrals for each of the three edges that make **up** that patch. This algorithm is quite efficient in a sequential code, but is not as appropriate for a parallel code where the three edges of both the source and testing patch (corresponding to the row and column indices of the matrix) will not generally be located in the same processor. The parallel version of PATCH currently only uses this calculation fOr those matrix elements that are local to the processor doing the computation introducing an inefficiency compared to a sequential algorithm. This inefficiency is specific to the integration algorithm used, and can be removed, for example, by communicating partial results computed in one processor to the processors that need them. This step has not been taken at this time in the parallel PATCH code, due partially to the complexity it adds to the code.

3.a.ii Matrix Equation Factorization and Solution

When the impedance matrix is assembled, the solution is completed by a LU factorization, scaling as N^3, and a forward and backward substitution used to solve for a given right hand side, scaling as N^2. The choice of the LU factorization algorithm determines the matrix decomposition scheme used on the set of processors.

One style of decomposition that is suitable for use with LINPACK [21] factorization routines is to partition the. matrix

$$A = \begin{pmatrix} A_0 \\ \vdots \\ A_{(p_* - 1)} \end{pmatrix} \qquad (8)$$

where $A_i \in C^{m_i \times N}$ and $m_i \approx N / P$. Each submatrix A_i is then assigned to processor i, where P is the number of processors. The LINPACK method involves many BLAS 1 (vector-vector) operations [22], which perform at 25 to 70 MFLOPS on the Cray T3D (150 MFLOPS peak performance.) These type operations perform poorly on the T3D and other hierarchical memory computers because the amount of work that is performed on the data brought from memory to the cache then to the processor is similar in size to the amount of that data.

Another type of decomposition is to assume that the physical processors, P_{ij} form a logical two-dimensional $p_r \times p_c$ array, where i and j refer to the row and column index of the processor. Then the simplest assignment of matrix elements to processors in a block method is to partition the matrix

$$A = \begin{pmatrix} A_{00} & \cdots & A_{0(p_c - 1)} \\ \vdots & \ddots & \vdots \\ A_{(p_r - 1)0} & \cdots & A_{(p_r - 1)(p_c - 1)} \end{pmatrix} \qquad (9)$$

where $A_{ij} \in C^{m_i \times n_j}$, $m_i \approx N / p_r$, and $n_j \approx N / p_c$. Submatrix A_{ij} is then assigned to processor P_{ij}. This is **an** appropriate decomposition for use with LAPACK routines [10] that use BLAS 3 (matrix-matrix) operations [23] and can perform at a high rate, typically 100 to 120 MFLOPS on a T3D processor. These operations perform a large amount of work on the data that has been brought into the cache, compared with BLAS 1 operations. However, this simple decomposition doesn't provide good load balance. It can be overcome by blocking the matrix into much smaller $k \times k$ submatrices, and *wrapping* these in two dimensions onto the logical processor array. In other words, partition

$$A = \begin{pmatrix} A_{00} & \cdots & A_{0(M-1)} \\ \vdots & \ddots & \vdots \\ A_{(M-1)0} & \cdots & A_{(M-1)(M-1)} \end{pmatrix}$$

(lo)

where $M \approx N/k$, and all blocks arc of size $k \times k$. Then block A_{ij} is assigned to processor $I'_{(i \ tip, \ )(j \ mod \ p_c)}$. This is the partitioning strategy that is used by PATCH. (On the T3D, $k = 32$ has been found to provide optimal results between load balance demanding the smallest possible k and the performance of the BLAS 3 operations requiring a large value for k.).

The PATCH code uses a matrix equation solver-narnti BSOLVE [24]---bascd on this deeomposition . Figure 1 show the total and per processor performance for BSOLVE for the largest matrix that can be solved on each size of machine. Total time scales at the same efficiency as total performance, and is 25 minutes for the 30,240 size matrix on 256 processors.
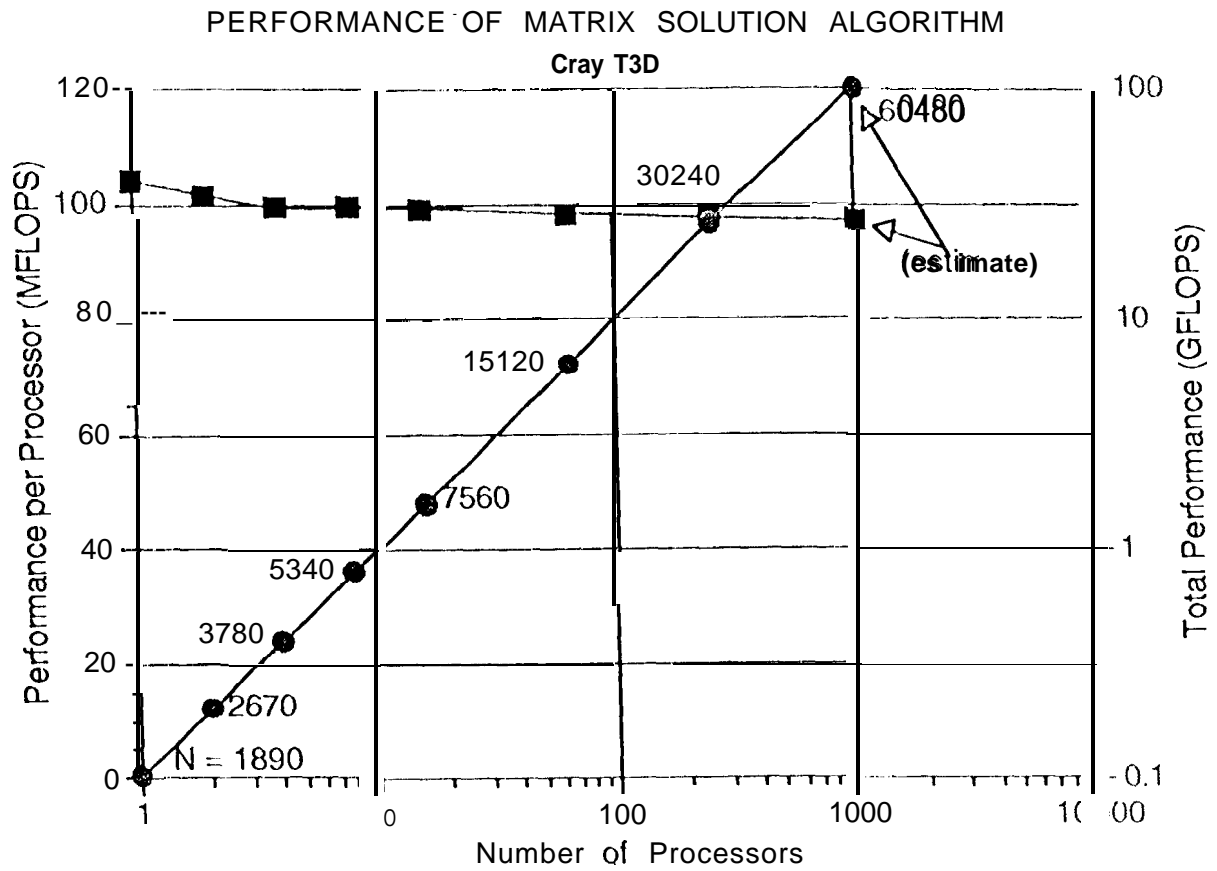
PERFORMANCE OF MATRIX SOLUTION ALGORITHM

Cray T3D

Figure 1. Performance for ^{B}SOLVE (includes factoring matrix, estimating condition number, and solving for one right hand side.)

Each processor of the T3D (60 Mbytes usable memory) can hold in memory a matrix piece of size 1890x 1890. The largest T3D has 1024 processors and can store a matrix of size 60,480 x 60,480 and can be factored in about 100 minutes. Problems in this class (those that are limited only by memory requirements and not by run-time requirements) arc good candidates for out-of-core methods. A larger matrix equation is stored on disk, and portions of the matrix arc loaded into memory and factored. For an out-of-core solution to be efficient, the work involved in loading part of a problem from disk must be overlapped with the work performed in solving the problem, so that storing the matrix on disk doesn't significantly increase the run time over what it would be on a machine with more memory. Specialized matrix equation solvers have been
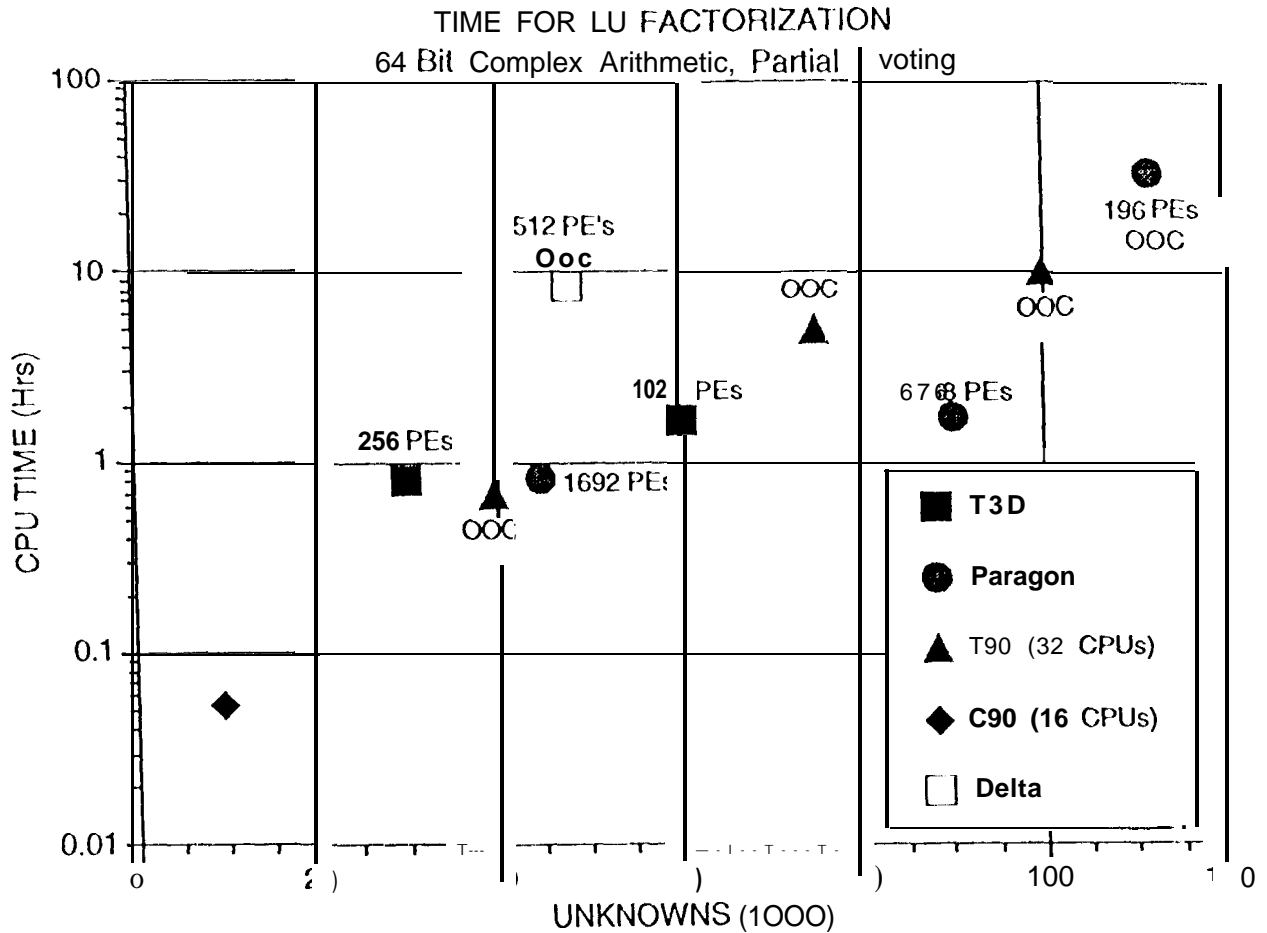
Figure 2. CPU time vs. number of unknowns for 64 bit complex dense direct solvers with partial pivoting for various machines (Cray T3D, T90 and C90; Intel Paragon and Delta). Processing elements (PEs) used in calculation shown for parallel computers. lo-core and out-of-core (OOC) results specified.

developed to efficiently exploit the large disc memory available on many parallel machines. Figure 2 shows general results for dense solvers, both in core and out of core, for various machines.

3.a.iii   Computation of observable

l-he observables of the PATCH code, such as radar cross section information or near or far field quantities, can be easily computed once the currents on the edges of the surface patches are known. These calculations involve forward integrals of the currents and the freespace Green's

13

function. Due to **the** discretized of this current, this integration results in a summation of field components duc to each current basis function. Since these currents arc distributed ove,all the processors in a parallel simulation, partial sums can be performed on the individual processors, followed by a global summation of these partial results to find **the** total sum. This calculation scale quite well since the only overhead is the single global sum.

3.a.iv Total Performance

Figure 3 shows overall code timings for a scaled size problem. Each case involves a matrix that fills the same fraction of each processor's memory.   As the problem size increases, more processors are used in the calculation.   It is clear that the matrix fill is dominant in the code, and in
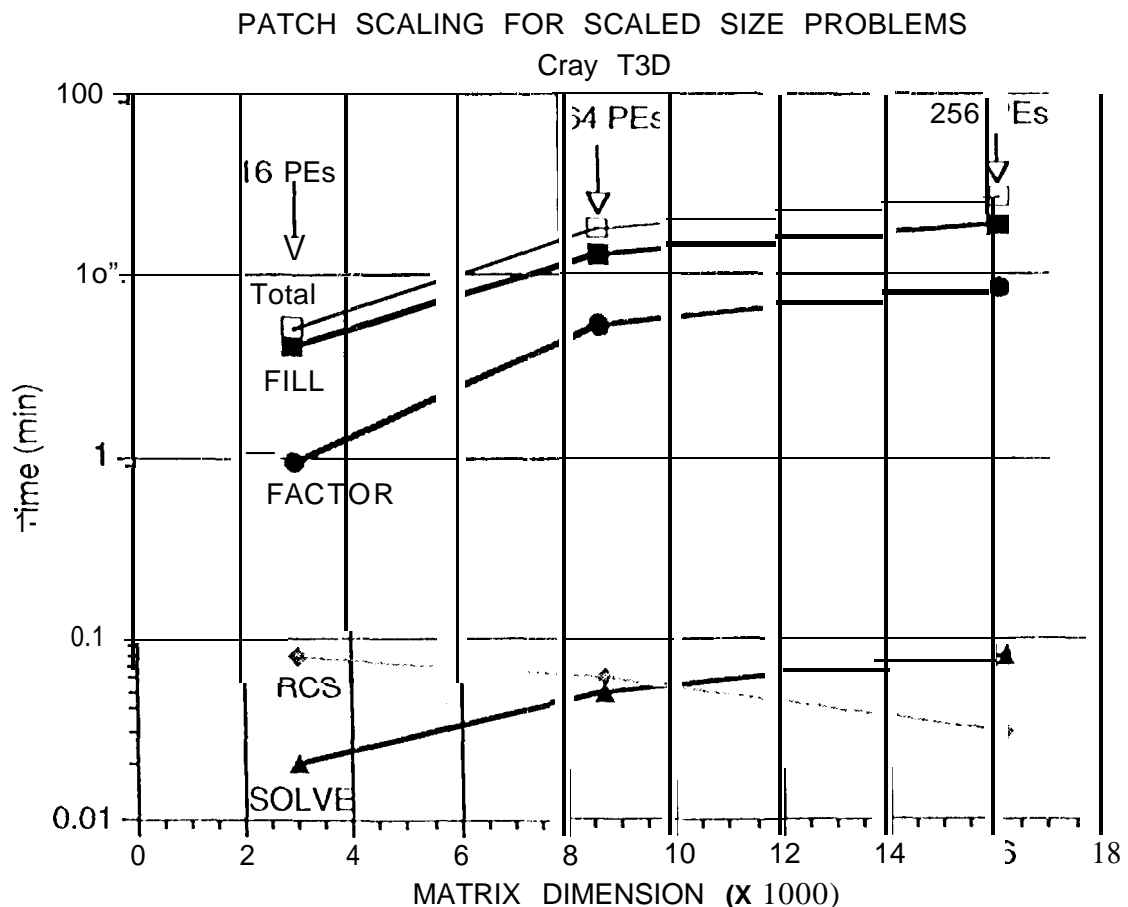
PATCH SCALING FOR SCALED SIZE PROBLEMS

Cray T3D

Figure 3. Scalability for I'Al'Cl I code for scaled size problems.

fact, the crossover point between the $O(n^3)$ factorization and the $O(n^2)$ fill has not been reached. Also, the time involved in the radar cross section calculation is shown to decrease **as** the number of processors is increased, as was described in the previous paragraph. It may also be observed that the matrix solve time, also $O(n^2)$, parallels the fill lime quite well.

Figure 4 shows scalings for the fixed size problems on the T3D. Each of the problems shown is run on the smallest set of processors needed to hold the matrix data, and then on larger numbers. The total code time initially decreases linearly with the number of processors, leveling off as the amount of communication time begins to become a larger fraction of the total time needed to complete the calculations.
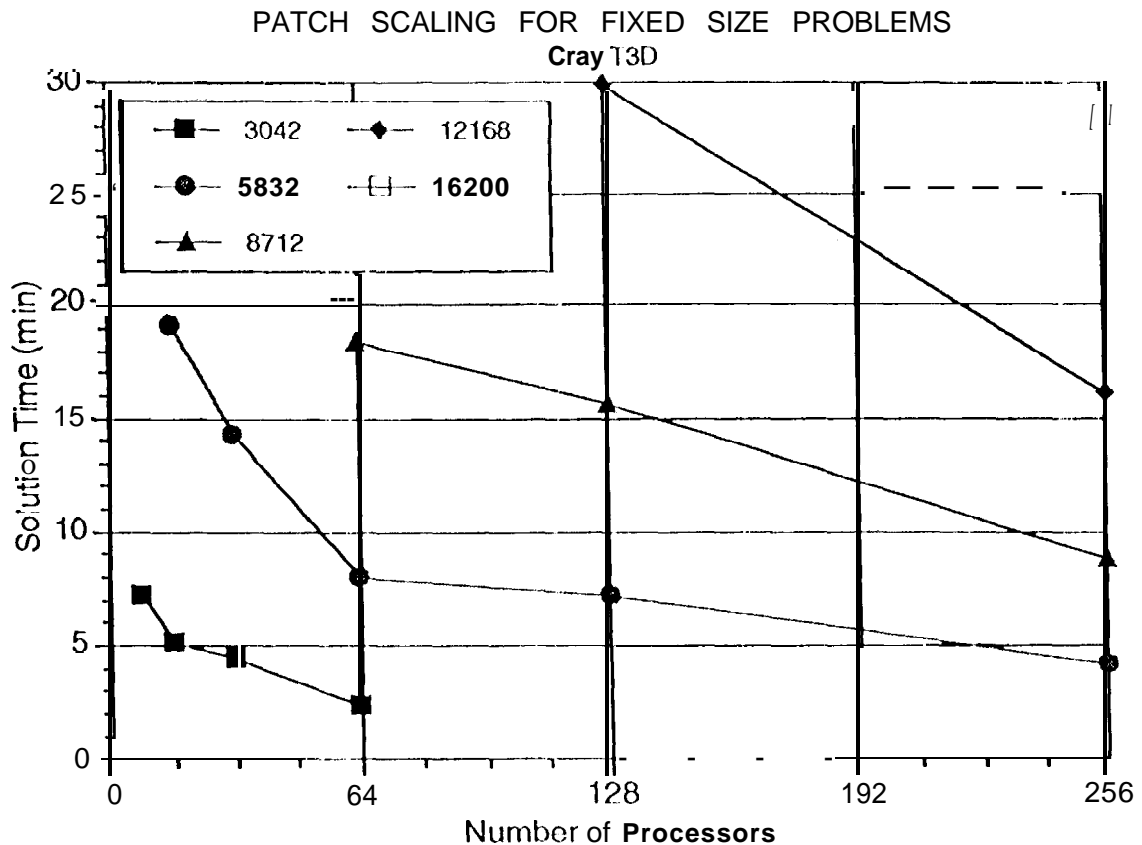
PATCH SCALING FOR FIXED SIZE PROBLEMS
Cray T3D

1 figure 4. Scalability y for PATCH code for fixed size problems. The 16,200 unknown case **was only run on 256 processors.**

4. FINITE ELEMENT FORMULATIONS

Volumetric modeling by (he use of an integral equation can also be used in simulations, though the available memory of current or planned technology greatly limits the size of problems that can be modeled. Because of (his limitation in modeling t}lrcc-dimensional space by integral equations, finite element solutions of the partial differential equations that lead to sparse systems of equations are common] y used [25]. A finite element model is natural when the problem contains inhomogenous material regions that surface integral equation methods are either incapable of modeling, or are very costly to model. The problem domain is broken into a finite element basis function set used to discretize the fields. The resulting linear system of equations—rather than scaling as the N^2 storage of the method of moments--scales as mN where m is the average number of non-zero matrix equation elements per row of the sparse 1 inear system. This value is dependent upon the order of the finite element used, but is typically between 10 and 100, and is independent of the size of the mesh. For a 6 unknown, vector edge-based tetrahedral finite element [26], m is typically 16.

Typicall y, the s ystem of equat ions resulting, from a finite element discretizat ion is symmetric; the non-zero structure of a representative example is shown in Figure 5 (left). A symmetric factorization of this system (Cholesky factorization) leads to [27]

$$A = \bar{L}D\bar{L}^T = \bar{L}D^{1/2}D^{1/2}\bar{L}^T = LL^T \tag{11}$$

where the diagonal matrix is specifically shown distributed symmetrically between the symmetric factors I., an important consideration when symmetrically applying an incomplete Cholesky preconditioner in iterative methods. The factorization results in non-zero elements in L where non-zeros exist in A, as well as fill-in, or new non-zero entries generated during the factorization. Fill-in requires additional storage for 1., as well as additional time to complete the factorization. To reduce the amount of fill-in, the system is reordered by applying a permutation

$$(PAP^T)PX = PB \tag{1 2.}$$

where the permutation matrix P satisfies $PP^T = 1$. In (12), PAP^T remains symmetric, and the forward and backward substitution phases become

$$\hat{L}Y = PB, \hat{L}^T Z = Y, X = P^T Z \qquad (13)$$

where $\hat{L}$ is the Cholesky factor of PAP^T. For a sparse factorization, the permutation matrix is chosen to minimize the amount of fill-in generated. Since there are $n!$ possibilities for P to minimize fill-in, heuristic methods are used to achieve a practical minimization, the most common being the minimum degree algorithm [28]. Figure 5 (right) shows the non-zero structure of the representative system after reordering for a canonical scattering problem.

"Table 2 lists scaling data for problem size when using a Cholesky factorization with the minimum degree reordering algorithm used to minimize storage. Based on the computer storage available, the number of edges in a edge-based tetrahedral mesh [18] along with the number of non-zeros in the factor $\hat{L}$ [29], and the volume that can be modeled is shown. The volume is based on the use of 15,000 tetrahedra per cubic wavelength, corresponding to approximately 25 edges per linear wavelength. 'I'his number can vary depending on the physical geometry (curvature, edges, points) and the local nature of the fields.

Theoretically, the system in (12) general] y is not positive definite, being symmetric indefinite, and a Cholesky factorization in this case is numerically unstable. For the practical solution of most problems, this has not been found to be a issue. Methods that preserve the
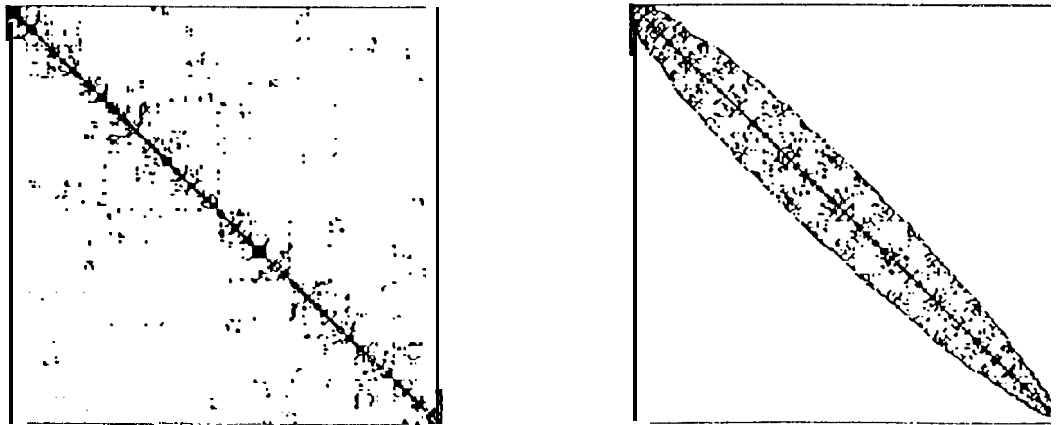

Figure 5. Non-zero matrix structure of typical finite element simulation. Left, original structure; right, structure after reordering to minimize bandwidth.

17

symmctric sparsity, and allow pivoting to produce mom stable algorithms can be used [28].

'J'able 2. Scaling of typical factorization finite clcmcnt matrix solution algorithms,

PROBLEM SIZE			
MEMORY (Mbytes)	N 10^3	$\|\hat{L}\|$ 10^6	VOL (λ^3)
128	30	8	2.0
256	50	16	3.3
512	82	32	5.5
1024	135	64	9.0
8,192	600	512	40
32,768	1,627	2,048	108
131,072	4,410	8,192	294

From 'l'able 2 it is seen that even though the storage for the finite clcmcnt method is linear in N, the fill-in duc to the usc of factorization algorithms causes the storage to grow as $N^{1.4}$. 1 .incar storage can be maintained using an iterative solution.   Sparse iterative algorithms for systems resulting from clcctromagnctic simulations recursively construct Krylov subspacc basis vectors that arc used to iteratively improve the solution to the linear systcm. The iterates are found from minimizing a norm of the residual

$$r = Ax - b \qquad (14)$$

at each step of the algorithm. (A single solution vector for a single excitation is shown.) Since the systcm is complex-valued indefinite, methods appropriate for this class of systcm such as bi-conjugate gradient, gcncralizcd minimal residual, and the quasi-minimal residual algorithm are applied [30]. They all require a matrix-vector multiply, and a set of vector inner products for the calculation. The iterative algorithms require the storage of the matrix and a fcw vectors of length N. When only the matrix and a fcw vectors nccd to bc stored, problems of very large size can bc handled, if the convergcncc rate is control] cd.   The number of itcrations (with a sparse matrix-dcnsc vector multiply accounting for over 90% of the time at each step of the iterative algorithm) dctcrmincs the time to solution.

Because the matrix-vector multiply **dominates the** KIYlov iterative methods, the algorithmic scaling is found from this operation. A single. sparse matrix-dense vector multiply requires mN operations, and if there arc I total iterations required for convergence, the number of floating point operations needed is $I * mN$. A typical solution of the system of equations, without the application of a preconditioner, may require a number of iterations $I = \sqrt{N}$, producing the squential algorithm scaling

$$mN^{3/2} \tag{15}$$

for the solution of a single right-hand-side. The algorithmic scaling for a direct factorization method is $O(m^2 N)$ [28, page 104]; therefore the factorization methods-—when memory is sufficient to hold the fill-in entries---give considerable CPU time savings in the solution. Further advantage is also gained over iterative methods when using a direct factorization. Modern computer architectures arc typically much less efficient at performing the sparse operations required in the sparse matrix-dense vector mutiply of the iterative algorithm, as compared to operations on dense matrix systems. Current direct factorization methods attempt to usc block algorithms, exploiting dense matrix sub-structure in the sparse system, and therefore increasing the performance of the factorization, further improving the performance from that given by the algorithmic scaling differences.

It is seen that the number of iterations directly increases the time to solution in the iterative methods. This number can be controlled to some degree by the usc of preconditioning methods that attempt to transform the matrix equation into onc with more favorable properties for an iterative solution. To control the convergence rate, the matrix A should be scaled by a diagonal matrix that produces ones along the diagonal. This scaling removes the dependence on different element sizes in a mesh. A preconditioner M can then be symmetrically applied to transform the system giving

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{M}^{-1}(\mathbf{M}\mathbf{x}) = \mathbf{M}^{-1}\mathbf{b}. \tag{16}$$

The right hand side vector is initially transformed, and the system is then solved for the intermediate vector $\bar{\mathbf{x}} = \mathbf{M}\mathbf{x}$, multiplying this vector by $\mathbf{M}^{-1}$, A and $\mathbf{M}^{-1}$ in succession at each

iterative step. When the solution has converged, **x** is recovered from x. The closer M is to 1, in (4), the quicker the transformed system will converge to a solution. A common preconditioner is an incomplete Cholesky factorization [31] where M is chosen as a piece of the factor L in (11). It is computed to keep some fraction of the true factorization elements, the exact number and sparsity location of the element dependent on the exact algorithm used. A useful form of incomplete factorization keeps the same number of elements in the incomplete factor as there are in A. This requires three times the number of operations at each iterative step; therefore the time to solution will be decreased if the number of iterations is lesser than one-third when applying this preconditioner.

When the right hand side consists of a number of vectors, newly developed block methods can be applied to the system to use the additional right hand sides to improve the convergence rate [32,33].

4.a Scalability on Parallel Computers

In a finite element algorithm, the resultant sparse system of equations is stored within a data structure that holds only the non-zero entries of the sparse system. This sparse system must ultimately be distributed over the parallel computer, requiring special algorithms to either break the original finite element mesh up into specially formed contiguous pieces, or by distributing up the matrix entries themselves onto the processors of the computer. As in the dense method of moments solution, the pieces are distributed in a manner that allows for an efficient solution of the matrix equation system.

4.a.i The Finite Element Mesh and the Sparse Matrix Equation

The volumetric region (V) is enclosed by a surface (∂V), *in which* a finite element discretization of a weak form of the wave equation is used to model the geometry and fields

$$-\frac{\eta_0}{jk_0} \iiint_V \left[\frac{1}{\varepsilon_r} (\nabla \times \bar{H}) \bullet (\nabla \times \bar{W}^*) - k^2 \mu_r \bar{H} \bullet \bar{W}^* \right] dv - \iint_{\partial V} \bar{E} \times \hat{n} \bullet \bar{W}^* ds = 0 \ . \qquad (17)$$

$\bar{H}$ is the magnetic field (the $\bar{H}$ -equation is used in this paper; a dual $\bar{E}$ -equation can also be written), $\bar{W}$ is a testing function, the asterisk denotes conjugation, and $\bar{E} \times \hat{n}$ is the tangential component of $\bar{E}$ on the bounding surface. In (17), ε_r and μ_r are the relative permittivity and permeability, respectively, and k_0 and η_0 are free-space wave number anti impedance, respectively. A set of finite clement basis functions, the tetrahedral, vector edge elements (Whitney elements) will be used to discretize (17),

$$\bar{W}_{mn}(r) = \lambda_m(r) \nabla \lambda_n(r) - \quad \mathbf{I \ v \ a \ n \ ,} \qquad (18)$$

where $\lambda(r)$ arc the tetrahedral shape functions and indices (m, n) refer to the two nodal points of each edge of the finite element mesh. These elements will be used for both expansion and testing (Galerkin's method) in the finite element domain. Because of the local nature of (17), (sub-domain basis functions and no Green's function involved in the integration of the fields), the system of equations resulting from the integration only contains non-zero entries when the finite elements overlap, or arc contiguous at an edge. Because the mesh is unstructured, containing elements of different size and orientation conforming to the geometry, the resultant matrix equation will have a sparsity structure that is also unstructured.

The sparsity structure is further altered by the form of the Sommerfeld boundary condition applied on the surface S. When local, s ymmetric absorbing conditions are applied on the boundary [34]-- entering into the calculation through the surface. integral in(17)- a matrix with the structure shown in Figure 5 (left) results. It is seen that the diagonal is entirely filled, corresponding to the self-terms in the volume integral in (17), with the m non-zero entries scattered along the row (or column) of the symmetric matrix. The location of these entries is completely dependent upon the ordering of the edges of the tetrahedral elements used in the discretization. If a different **shape or** order of the elements arc used, the non-zero structure will differ slightly from the one shown. When an integral equation method is used to truncate he mesh [35,36], a dense block of elements

will appear in the lower right of the system (when the edges of the finite element mesh on the boundary are ordered last) as shown in Figure 6 (left). The integral equation approach to truncating the mesh uses the finite element facets on the boundary as source fields in an integral equation, resulting in a formulation for this piece of the calculation similar to that in Section 3, and with an amount of storage needed for the dense matrix as a function of the electrical surface area tabulated in Section 2. To circumvent the large dense storage needed with this application of a global boundary condition, a surface of revolution can be used to truncate the mesh [37,38], using a set of global basis functions to discretize the integral equation on this surface. This results in a system similar to that in Fig 6 (right) containing very small diagonal blocks due to the orthogonal global basis functions along the surface of revolution truncating the mesh, as well as a matrix coupling the basis functions in the integral equation solution to the finite element basis function on the surface. These coupling terms are the banded thin rectangular matrices symmetric about the diagonal of the matrix. Other forms of the integral equation solution, as outlined in Section 3, can also be used to discretize the integral equation modeling fields on the mesh boundary, leading to slight variations of the matrix systems shown in Figure 6.
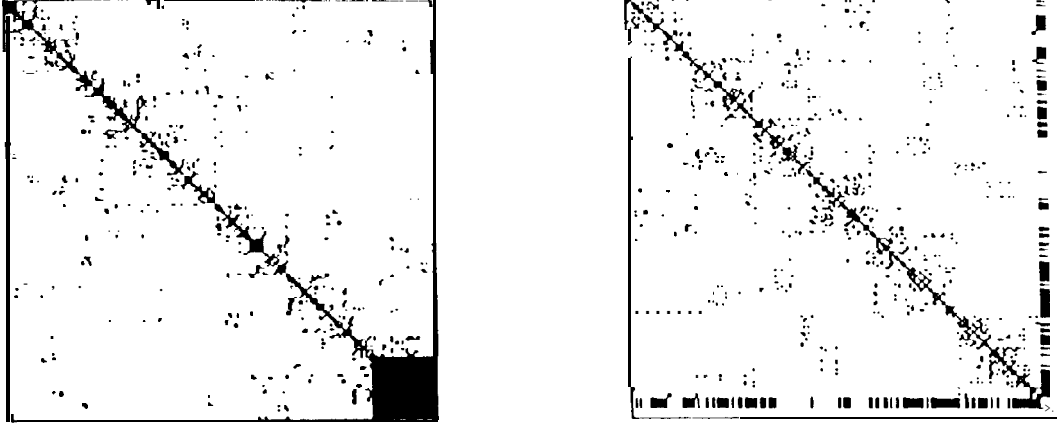
Figure 6. Non-zero matrix sparsity structure for system with dense surface integral equation boundary condition applied (left), and surface of revolution integral equation boundary condition (right). The mesh has 5,343 edges with 936 of those on the boundary.

The systems graphically represented in Figure (6) generall y have the form

$$\begin{bmatrix} K & C \\ C^\dagger & Z \end{bmatrix} \begin{bmatrix} H \\ I \end{bmatrix} = \begin{bmatrix} 0 \\ V \end{bmatrix} \qquad (19)$$

where K is the sparse, symmetric finite clement matrix, found from the volume integral in (17), C can be termed the coupling matrices that arc interactions between the finite elements at the boundary and the integral equation basis functions, and Z represents the integral equation, method of moments matrix entries. The symbol $\dagger$ indicates the adjoint of a matrix. H is the vector of magnetic field coefficients for each finite element, and I represents the equivalent current basis functions on the boundary of the mesh. For a scattering problem formulation, the incident field couples only to the integral equation boundary, and is represented as V. For radiation problems the O and V vectors arc interchanged since the impressed source is modeled in the **mesh. I**)iffering formulations lead to variations in (19), but the general algebraic nature is preserved. To exploit the sparsit y of K in (19), the system is solved in two steps by initially substituting $H = -K^{-1}CI$ from the first equation in (19) into the second, producing

$$(Z - C^\dagger K^{-1}C)I = V . \qquad (20)$$

This system has a size on the order of the number of basis functions in the integral equation **model,** is dense, and can be solved by either direct factorization or iterative means as outlined in Section 3. The intermediate calculation, $KX = C$ is the sparse system of equations to be solved, producing X.

The solution of this sparse system on a parallel computer requires it **to be** distributed. Traditionally, the dependence between mesh data and the resultant sparse matrix data is exploited in the development of mesh partitioning algorithms [39-42,55]. These algorithms break the physical mesh or its graph into contiguous pieces that are then read into each processor of a distributed memory machine. The mesh pieces are generated to have roughly the same number of finite elements, and to some measure, each piece has minimal surface area. Since the matrix assembly routine (the volume integral in (17)) generates non-zero matrix entries that correspond to the direct interconnection of finite elements, the mesh partitioning algorithm attempts to create a load balance of the sparse system of equations. Processor communications in the algorithm that solves the sparse system is limited by the abi 1 it y to minimize the surface area of each mesh piece.

Mesh partitioning algorithms are generally divided into multilevel spectral partitioning, geometric partitioning, and multilevel graph partitioning. Spectral partitioning methods [40,4 1] creates eigenvectors associated with the sparse matrix, and uses this information to recursively break the mesh into roughly equal pieces. It requires the mesh connectivity information as input, and returns lists of finite elements for each processor. Geometric partitioning *[39]* is an intuitive procedure that divides the finite element mesh into pieces based on the geometric (node x, y, z coordinates) of the finite clement mesh. This algorithm requires the mesh connectivity as well as the node spatial coordinates and returns lists of finite elements for each processor. Graph partitioning [42] operates on the graph of the finite element mesh (mesh connectivity information) to collapse (or coarsen) vertices and edges into a smaller graph. This smaller graph is partitioned into pieces, and then uncoarsened and refined for the final partitions of finite elements for the parallel processors. The input and output is identical to spectral methods. Multilevel algorithms **operate** by performing multiple stages of the partitioning simultaneously, accelerating the

algorithm. Most of the algorithms and their offshoots perform similarly in practice, with the spectral and graph partitioning algorithms being simpler to use since they do not need geometry information. An alternative to these mesh partitioning algorithms is a method that divides the matrix entries directly, without operating on the finite element mesh, and will be examined in Section 4.a. iv.

Different decompositions are used depending whether direct factorization or iterative methods are used in the solution. Decompositions for iterative solutions, as well as the iterative methods themselves have shown greater case in parallelization than direct factorization methods. Both approaches will now be considered.

4.a.ii Direct Sparse Factorization Methods

Direct factorization methods require a sequence of four steps; reordering of the sparse system to minimize fill-in, a symbolic factorization stage to determine the structure and storage of $\hat{L}$ in (13), the numeric factorization producing **the** complex-valued entries of $\hat{L}$, and the triangular forward and backward solutions. The fundamental difficulty in the parallel sparse factorization is the development of an efficient reordering algorithm that minimizes the fill-in and will scale well on distributed memory machines, controlling the amount of communication necessary in the computation. The minimum degree algorithm typically used in sequential packages is inherently non-parallel, proceeding sequentially in the elimination of nodes in the graph representing the non-zero structure of the matrix. Other algorithms for reordering, as well as the following symbolic and numeric factorization steps that depend on this ordering are under study [43]. Current factorization algorithms [44,45,46] can exhibit fast parallel solution times on moderately large sized problems, but are dependent on the relative structure of the mesh, whether or not the problem is two or three-dimensional, and the relative sparsity of the non-zero entries. For problems with more structure and less sparsity, higher performance is found by these sparse factorization solvers.

4.a.iii Sparse Iterative Solution Methods

1 'or a parallel implementation of the sparse iterative solvers introduced above, a decomposition of the matrix onto the processors that minimize communication of the overlapping vector pieces in the parallel matrix-vector multiply of the iterative algorithm, reduces storage of the resultant dense vector pieces on each processor, and allows for load balance in storage and computation is required. Various parallel packages have been written that accomplish these goals to some degree [47,48]. The mesh decompositions outlined previously can be used and integrated with the parallel iterative algorithm to solve the system.

Ah-natively, a relatively simple approach that divides the sparse matrix entries among the distributed memory processors ean be employed **[49].** The matrix is decomposed in this implementation into row slabs of the sparse reordered system. The reordering is chosen to minimize and equalize the bandwidth of each row over the system [17,18] (as shown in Fig .5 right) since the amount of data communicated in the matrix-vector multiply will depend upon the combination of equalizing the row bandwidth as well as minimizing it. A row slab matrix decomposition strikes a balance between near perfect data and computational load balance among the processors, minimal but not perfectly optimal communication of data in the matrix-vector multiply operation, and scalability of simulating larger sized problems on greater numbers of processors. Since the right-hand-side vectors in the parallel sparse matrix equation ($KX = C$) are the columns of C, these columns are distributed as required by the row distribution of K. When setting up the row slab decomposition, K is split by attempting to equalize the number of non-zeros in each processor's portion of **K** (composed of consecutive rows of **K). The rows** in a given processors portion of **K** determines the rows of **C** that processor will contain. As an example, if the total number of non-zeros in **K** is nz , a loop over the rows of **K** will be executed, counting the number of non-zeros of **K** in the rows examined. When this number becomes approximately nz / P (where P is the number of processors that will be **used** by the matrix equation solver), the set of rows of K for a given processor has been determined, as has the set of tows of C .

The matrix decomposition code used in this example consists of a number of subroutines; initially, the potentially large mesh files arc read (READ), then the connectivity structure of the sparse matrix is generated and reordered (CONNECT), followed by the generation of the complex-valued entries of K (FEM), building the connectivity structure and filling the C matrix (COUPLING). Finally the individual files containing the row slabs of K and the row slabs of C must be written to disk (WRITE). For each processor that will be used in the matrix equation solver, one file containing the appropriate parts of both the K and C matrices is written. Figure 7 shows the performance of these routines over varying numbers of processors for a problem simulating scattering from a dielectric cylinder modeled by 43,791 edges. The parallel times on a Cray T3D arc compared against the code running sequentially on one processor of a 190. As mentioned above, the reordering algorithm and the algorithm generating the matrix connectivity arc fundamentally sequential. These routines do not show high efficiency when using multiple processors-the time for this algorithm is basically flat--whereas for routines that can be parallelized (FEM, COUPLING and WRITE), doubling the number of processors reduces the amount of time by a factor of approximately two. The time for reading the mesh is bound by I/O rates of the computer, and the time for writing the decomposed matrix data varies slightly for the 128 and 256 processor cases due to other users also doing I/O on the system. As will be shown in the next result, a key point of this approach to matrix' decomposition is that the total time needed (less than 100 sec. on 8 processors) is substantially less than the time needed for solving the linear system, and any inefficiencies here are less important than those in the iterative solver.
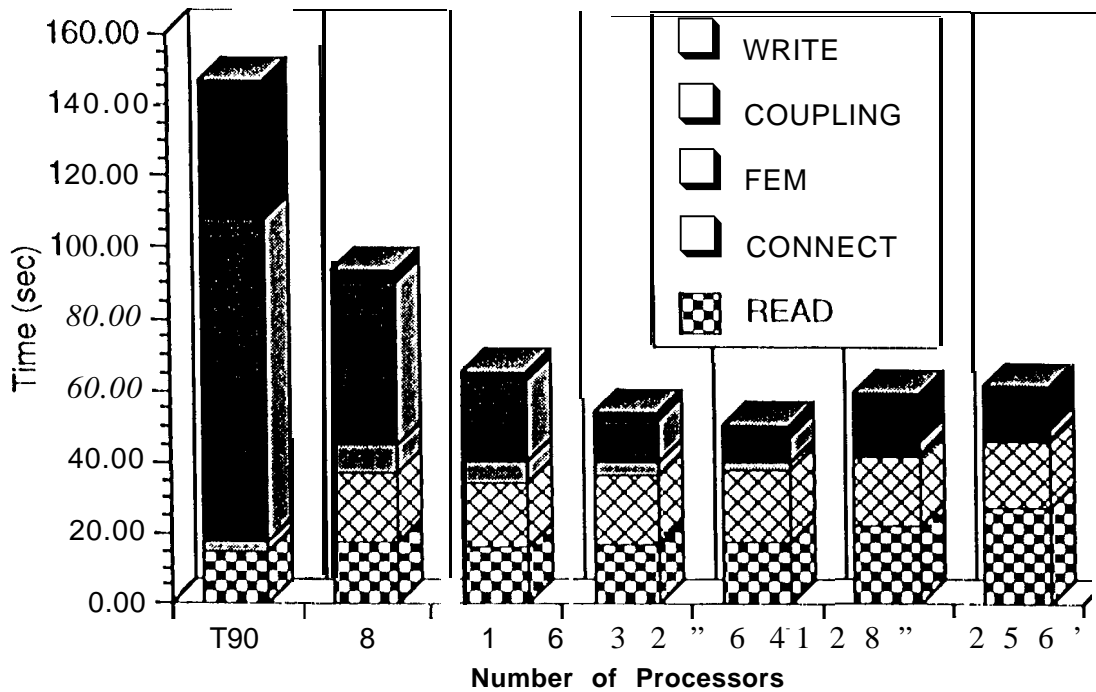
SCALING OF SPARSE SLICE ALGORITHM

1 'igurc 7. Computation time and scaling for a relatively small simulation (dielectric cylinder with 43,791 edges, radius = 1 cm, height = 10 cm, permittivity = 4.0 at 5.0 GIIz). First column shows time for single processor T90. Times on T90 for CONNECT and FEM have been combined.

In this example, quasi-minimum residual algorithm [52] is used to solve the sparse systcm of equations $KX = C$. With the row slab dccomposition used, the machine is logically considered to bc a linear array of processors, with each slab of data residing in one of the processors. Central components of the quasi-minimum residual algorithm that arc affcctcd by the usc of a distributed memory machine arc the parallel sparse matrix- dense vector multiply, and dot products and norm calculations that need vector data distributed over the machine. The dominant component is the matrix- vcctor multiply, accounting for approximately 80% of the time required in a solution. The parallel sparse matrix-dense vector multiply involves multiplying the K matrix that is distributed
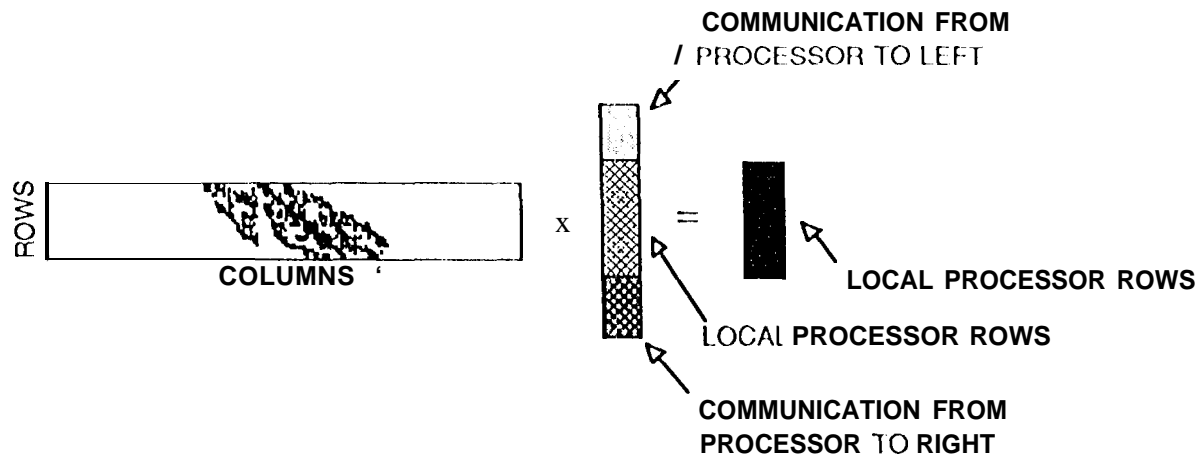
2.8

Figure 8. Local sparse ma(rix-dense vector multiply graphically displayed.

across the processors in row slabs, each containing a roughly equal number of non-zero elements, and a dense vector x, that is also distributed over the processors, to form a product vector y, distributed as is x (Figure 8). Since the K matrix has been reordered for minimum bandwidth, the minimum and maximum column indices of the slab arc known. If the piece of the dense vector x local to this processor has indices within this extent of column indices, the multiply may be dcmc locally and the resultant vector y will be purely local. In general, the local row indices of the dense vector x do not contain the range of column indices; therefore a communication step is required to obtain the portions of the multiply vector x required by the column indices of the **K** matrix, This communication step only requires data from a fcw processors to the left and right. The exact number of processors communicating data is dcpcndcnt on the row bandwidth of the local piece of **K,** and the number of processors being used. In the simulations considered, the number of processors communicating data is typically onc or two in each direction on scaled problems.

Shown in Figure *9 arc* plots of time to convergence on different numbers of processors for five different problems (fixed size problems). The number of unknowns in the finite element mesh and the number of columns of C, arc indicated on the plots. The quasi-minimum residual algorithm was stopped when the normalized residual was reduced three orders of magnitude for each column of C. With an initial guess being the zero vector, this results in a normalized residual

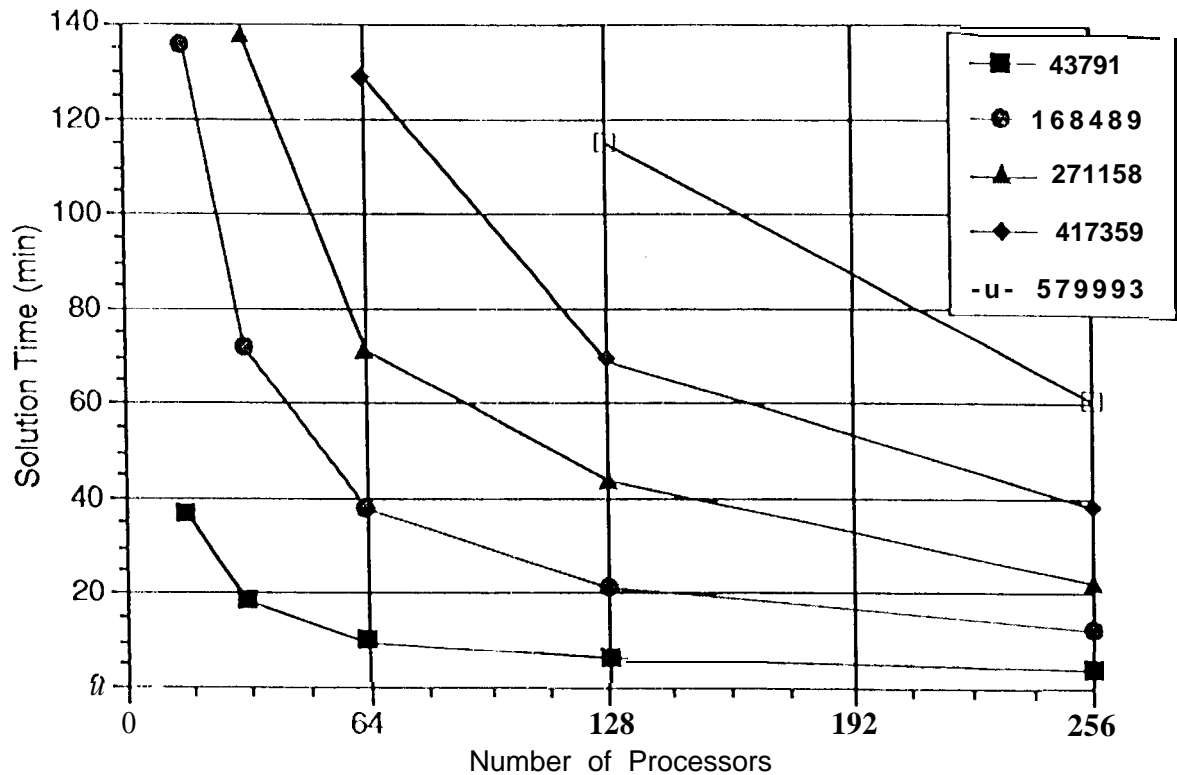**ITERATIVE SOLUTION SCALING FOR FIXED SIZE PROBLEMS**

Figure 9. Time of convergence for five different problems. The time
shown is the total execution time for the solver on different numbers of
processors. The **C** matrix had 116 columns in each case.

of 0.1 %, a value that is sufficient for this scattering problem. Given a fixed communication
percentage and a fixed rate for local work, doubling the number of processors for a given problem
would halve the total solution time. The curves in Figure **9** do not drop linearly at this rate for
increasing numbers of processors, because there is a decrease in the amount of work per processor
while the amount of data communicated increases, causing tbc curves to level off.

Another factor in the performance of the parallel matrix-vector multiply is the percentage of
communication. This is related to the number of processors to the left and right that each processor
must communicate. It is clear that running a fixed size problem on an increasing number of
processors will generate a growing amount of communication. The amount of communication is a

function of how finely the K matrix is decomposed, since its maximum row bandwidth after reordering is not a function of the number of processors used in the decomposition. If the maximum row bandwidth is m and each processor in a given d~.opposition has approximately m rows of K, then most processors wi ll require one processor in each direction for communication If the number of processors used for the distribution of K is doubled, each processor will have. approximatel y m/2 rows of K. Since the row bandwidth doesn' t change, each processor will now require communication in each direction from two processors. But since the number of floating point operations required hasn't changed, the communication percentage should roughly double. This can be seen in Figure 10, which shows communication percentage versus number of processors, for four problem sizes.
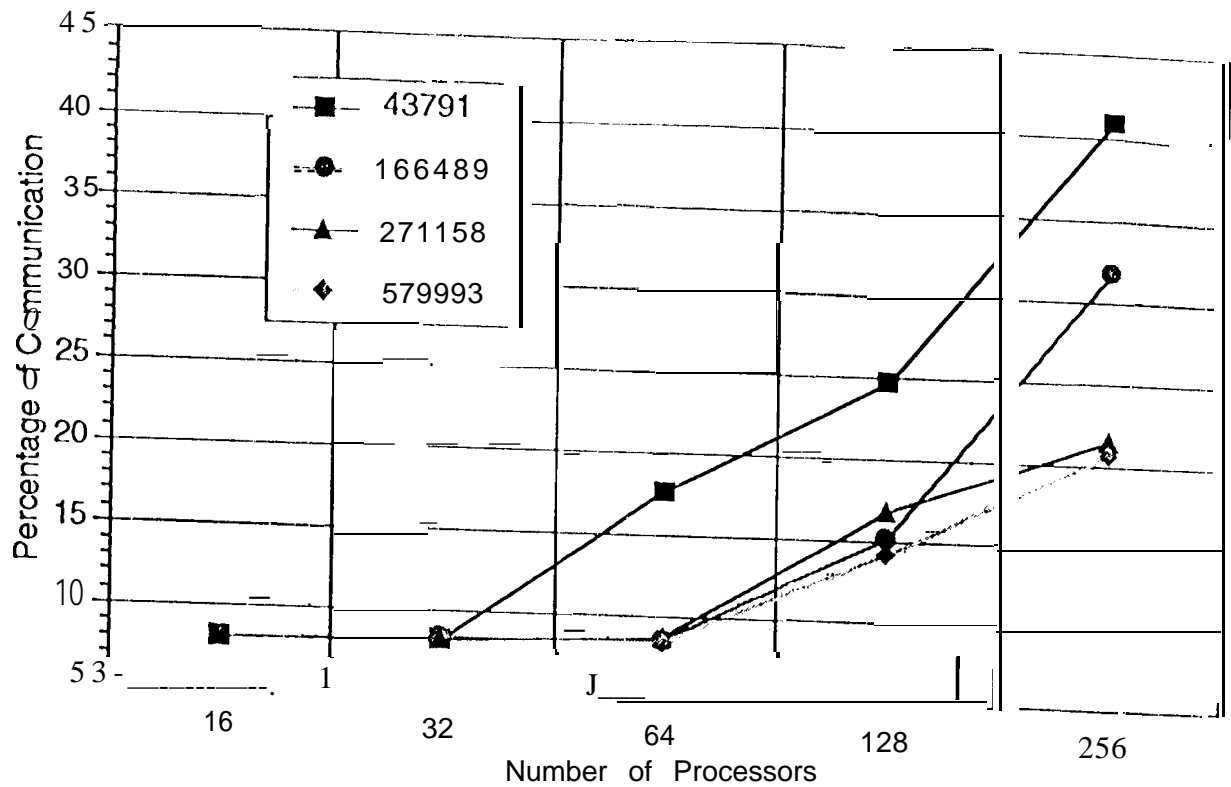
Figure 10. Percentage of communication versus number of processors for parallel matrix vector multiply, for four different size (number of edges) meshes of dielectric cylinder.

The row slab decomposition is a simple means for breaking the sparse matrix equation among the processors and while the mesh decomposition algorithms outlined above can also be used, differences between the approaches in time to solution on a parallel computer were found to be small for either approach. Two alternative mesh decomposition schemes have been compared to the matrix partitioning algorithm, contrasting data load balance, communication load balance, the total amount of communication and the performance of the local processor matrix-vector performance resulting from the specific decomposition used. The first is an algorithm termed JOSTLE [S5] that uses various optimization methods to equalize the mesh partitions among the processors. The second is a multi-level graph partitioning scheme termed METIS [42]. Among

the three approaches, no discernable difference was found in data and communication load balance, and in the performance of the local processor matri x- vector performance. A difference was found in the total amount of communciation needed in the solution of the sparse system of equat ions. When normalizing the total amount of communication in the matrix partition algorithm to 1.0, the JOSTLE algorithm reduced the communication overhead to 0.26, and the METIS algorithm reduced it to 0.22. From Figure 10, it is noted that the percentage of communication time in the complete solver is 8% for scaled-sized problems (those that fit into the minimal number of processors needed to solve the problem). It is this fraction of the total CPU time that can be reduced by the 0.26 and 0.22 fractions found using the mesh decomposition algorithms; i .e. the total time to solve the system would be reduced by just over 6% using the METIS algorithm for mesh decomposition. It was found that the METIS and JOSTLE algorithms did produce less communication overhead as the fixed size problem was solved on larger numbers of processors, thereby further reducing total exection time. This savings over the matrix partitioning method is offset though, since the overall execut ion time decreases dramatically y as seen in Figure 9 for a fixed size problem.

Krylov subspace methods different from the quasi-minimum residual algorithm can be coupled with a mesh or matrix decomposition method and used for sparse matrix solution. In [4'7] the conjugate gradient squared and generalized minimum residual method arc used with geometric partitioning algorithms and then compared. The Krylov iterative method implementations are necessarily similar since the dominant component of the solver is the matrix-vector multiply. Parallel speedup for fixed sized problems arc reported in [47] for the conjugate gradient squared and the generalized minimum residual method. The speedups arc very similar to those shown in Figure 9.

A possible means to substantially shorten the solution time in an iterative solution is the use of an effective preconditioner. The usc of incomplete Cholesky preconditioners used in sequential calculations is difficult to implement in a distributed memory parallel environment due to the need for performing a forward and backward solution with each matrix multiply step in (16). On a

parallel machine, these arc essentially sequential operations that can give greatly reduced performance [53]. A promising alternative is to calculate an approximation to the invers$_e$ of the system, rather than a factorization of the system as is done in the incomplete Cholesky approximation. A sparse approximate inverse [54] produces a matrix with a controllable number of non-zeros that approximates the inverse of K, and rather than calculating forward and backward solutions, it multiplies K at each step of the iterative algorithm. The matrix-matrix multiply can be achieved with much higher performance than the forward and backward solutions used in the incomplete factorization.

S. DISCUSSION

This paper presented an overview of solutions to surface integral equation and volumetric finite element methods on sequential and distributed memory computer architectures. Both the sequential algorithmic scalability as well as scalabilty on parallel computer systems were presented for current computer technology, with extrapolation to next generation technologies. A broad set of references arc given. When a uniform resource locator (URL) is also referenced, it points to software which is freely available.

REFERENCES

[1] E. Miller, "A selective survey of computational electromagnetics," *IEEE Trans. Antennas Propag.* AP-36, pp. 1281-1305, 1988.

[2] M. N. O. Sadiku, *Numerical Techniques in Electromagnetics*, Boca Raton: CRC Press, 1992.

[3] J. J. Dongarra, L. Grandinetti, G. R. Joubert, J. Kowalik, Editors, *High Performance Computing: Technology, Methods and Applications,* Advances in Parallel Computing 10, Amsterdam: Elsevier, 1995.

[4] E. Miller, "Solving bigger problems- by decreasing the operation count and increasing the computation bandwidth," *Proc. IEEE*, Vol. 79, *No. 10,* pp. 1493-1504, 1991.

[5] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing,* Redwood City: The Benjamin/Cummings Publishing Company, 1994.   See also http://www.cs.umn.edu/~kumar.

[6] "1". Cwik and J. Patterson, Editors, *Computational Electronics and Supercomputer Architecture,* Progress in Electromagnetics Research, Vol. 7, Cambridge: EMW Publishing, 1993.

[7] R. F. Harrington, *Field Computation by Moment Method. New* York: Macmillan, 1968.

[8] A. J. Poggio and E. K. Miller, "Integral equation solutions of three-dimensional scattering problems," in *Computer Techniques for Electromagnetics,* R. Mittra, Editor, Chapter 4, New York: Hemisphere, 1973.

[9] G. Golub and C. Van ban, *"Matrix Computations,"* The John Hopkins University Press, 1989.

[10] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S, Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson, *LAPACK User's Guide,* Society for Industrial   and   Applied   Mathematics,   Philadelphia. PA   1992.   See   also http://nct lib2.cs.utk.cdu/ lapack/index.ht ml

[11] E. Yip and B. Dembart, "Monostatic calculations for a 3D MOM code with fast mutipole method," *Progress in Electromagnetics Research Symposium Proceedings,* p. 53, Jul y 24-28, 1995.

[12] J. Rahola, "Solution of dense systems of linear equations in the discrete-dipole approximation, " *SIAM J. Sci. Comput.,* Vol. 17, No. 1, pp. 78-89, 1996.

[13] M. Sancer, R. McClary, K. Glover, "Electromagnetic computation using parametric geometry," *Electromagnetics,* Vol. 10, No. 1-2, pp. 85-104, 1990.

[14] F. X. Canning, "The Impedance Matrix Localization (IML) Method for Mon~ent-Method Calculations," *IEEE Antennas Propag. Mag., pp. 18-30,* October 1990.

[15] W. Chew, C. Lu and Y. Wang, "Efficient computation of three-dimensional scattering of vector electromagnetic waves," *J. Opt. Soc. Am. A,* Vol. 11, No. 4, pp. 1528-1537, 1994.

[16] V. Rokhlin, "Rapid Solution of integral equations of scattering theory in two dimensions," *J. Comp. Physics 86,* pp. 414-439, 1990.

[17] R. Coifman, V. Rokhlin and S. Wandzura, "The fast multipole method for the wave equation: a pedestrian prescription," *IEEE Ant. Propag. Msg. 35,* pp. *7-12,* **1993.**

[18] M. A. Stalzer, "A parallel fast multipole method for the Helmholtz equat i on," *Parallel l'recessing Letters 5,* pp. *263-2'14, 1995.*

[19] *S. M.* Rae, D. R. Wilton, and A.. W. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Trans. Antennas Propag.* AP-30, pp. 409-418, 1982.

[20] W. Johnson, D. R. Wilton, and R. M. Sharpe, "Modeling scattering from and radiation by arbitrary shaped objects with the electric field integral equation triangular surface patch code," *Electromagnetics* 10, pp. 41-64, 1990.

[21] J. J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, *LINPACK User's Guide,* Society for Industrial and Applied Mathematics, Philadelphia, PA 1979. See **also** http://netlib2.cs.utk.edu/linpack/index.html.

[22] J. J. Dongarra, J. Du Croz., S. Hammarling, and R. J. Hanson, An Extended set of FORTRAN Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.,* Vol. 14, pp. 1-17, 1988. See also http: //nctlib2.cs.utk. cdu/blas/indcx. htn~l.

[23] J. J. Dongarra, J. Du Croz, 1. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.,* Vol. 16 pp. 1-17, 1990. SeC also ht[p://nctlib2. cs.utk.cdu/blas/indcx .htnd.

[24] T. Cwik, R. van de Geijn, and J. Patterson, "Application of massively parallel computation to integral equation models of electromagnetic scattering," *J. Opt. Sot. Am.* 11, pp. 1538-1545 (1994). See also ftp://microwavc.jpl .nasa.gov/pub/PARAI .I.I3I /COMPLEX.SOLVER.

[25] J. Jin, The *Finite Element in Electromagnetics,* John Wiley and Sons, Inc., New York, 1993.

[26] J. Lee, and R. Mittra, "A Note On The Application Of Edge-Elements For Modeling 3-Dimensional Inhomogeneously- Filled Cavities," *IEEE Transactions On Microwave Theory and Techniques, vol. 40, no. 9, pp. 176"1- 1773, Sept., 1992.*

[27] S. Pissanetzky, *Sparse Matrix Technolgy*, London: Academic Press, 1984.

[28] I. S. Duff, A. M. Erisman and J. K. Reid, *Direct Methods for Sparse Matrices,* New York: Oxford University Press, 1986.

[29] E. Ng, and B. Peyton, "Block sparse Cholesky Algorithms on Advanced Uniprocessor Computers," *SIAM J. Sci. Comput.*, Vol.14, No. 5, pp. *1034-1056,* pp. 1034-1056, 1993.

[30] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods,* Society for Industrial and Applied Mathematics, Philadelphia. PA 1994. See also http://netlib2.cs.utk.edu/templates/index.html.

[31] M. Jones and P. Plassmann, "An improved incomplete Cholesky Factorization," *ACM Trans. Math. Software*, Vol. 21, No. 1, pp. 5-17, 1995. See **also** http://netlib2.cs.utk.edu/toms/740.

[32] D. O'Leary, "Parallel implementation of the block conjugate algorithm," *Parallel Comput.,* Vol. 5, pp. 127-139, 1987

[33] R. Freund, and M. Malhotra, "A block-QMR algorithm for non-hermitian linear systems with multiple right-hand sides," preprint 1996.

[34] R. Mittra, O. Ramahi, A. Khebir, R. Gordon, and A. Kouki, "A Review of Absorbing Boundary Conditions for Two- and Three-Dirnensional Electromagnetic Scattering Problems," *IEEE Trans. Mag, VOL* 25, no. 7, pp. 3034--3040, July 1989.

[35] J.-M. Jin and V. Liepa, "Application of Hybrid Finite Element Method to Electromagnetic Scattering from Coated Cylinders," *IEEE Tram. Antennas and Propagation, vol. AP-36, pp. 50--54,* Jan. 1988.

[36] X. Yuan, D. Lynch, and J. Strohbehn, "Coupling of finite element and moment mehods for electromagnetic scattering from inhomogenous objects," *IEEE Trans. Antennas Propagat., vol. AP-38*, pp. 386–394, Mar. 1990.

[37] W. Boyse and A. Seidl, "A Hybrid Finite Element Method for Near Bodies of Revolution," *IEEE Trans. Mag, vol.* 27, pp. 3833--3836, Sept. 1991.

[38] '1'. Cwik, C. Zuffada, and V. Jamnejad, "Modeling Three-Dimensional Scatterers Using a Coupled Finite Element–Integral Equation Representation," *IEEE Trans. Antennas Propag.*, AP 44, pp. 453-459, 1996.

[39] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving Finite Element Equations on Concurrent Computers," *American* Sot. *Mech. Eng.,* A. Noor Editor, pp. 291-307, 1986.

[40] A. Pothen, H. Simon and K. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM J. Matrix Anal. Appl.*, *vol.* 11, pp. **430-452, 1990.**

[41] B. Hendrickson and R. Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations, " *SIAM J. Sci. Comput.,* vol. 16, pp. 452-469, 1995.

[42] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *Technical Report TR 95-035,* Department of Computer Science, University of Minnesota, 1995. See also http: //www.cs.umn.edu/~karypis/metis/metis.html.

[43] M. Heath, E. Ng, and B. Peyton, "Parallel algorithms for sparse linear systems," *SIAM Review,* Vol. 33, No. 3, pp. 420-460, 1991.

[44] E. Rothberg, "Alternatives for solving sparse triangular systems on distributed-memory multi-processors," *Parallel Comput.,* Vol. *21,* pp. *1121-1136, 1995.* See also http: //www.ssd.intel.com/appsw/ses.html.

[45] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization," *Technical Report TR 94-63,* Department of Computer Science, University of Minnesota, 1994

[46] G. Hennigan, W. Dearholt, S. Castillo, "The finite element solution of elliptic systems on a massively parallel computer using a direct solver," preprint.

[47] J. Shadid and R. Tuminaro, "Sparse iterative algorithm software for large-scale MIMD machines: an initial discussion and implementation," *Concurrency: Practice and Experience,* Vol. 4, pp. 481-497, 1992. See also http://www.cs.sandia.gov/HPCCIT/aztec.html.

[48] M. Jones **and P.** Plassman, **"Scalable iterative** solutions of sparse linear systems," *Parallel Comput.,* vol. 20, pp.753-773, 1994. See also http://www.mcs.anl.gov/home/freitag/SC94demo/software/blocksolve.html.

[49] T. Cwik, D. Katz, C. Zuffada, and V. Jamnejad, "The Application of Scalable Distributed Memory Computers to the Finite Element Modeling of Electromagnetic Scattering," submitted to *Intl. J. Numer. Methods Engr., 1996.*

[50] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite* Systems, Prentice Hall, New Jersey, 1981.

[51] J. Lewis, "Implementation Of The Gibbs- Poole-Stockmeyer And Gibbs-King Algorithms," *ACM Trans. On Math. Software,* vol. 8, pp. 180-189, 1982. http: //netlib2.cs.utk. edu/toms/582.

[52] R. Freund, "Conjugate Gradient-Type Methods for Linear Systems with Complex Symmetric Coefficient Matrices," *SIAM J. Stat. Comput, vol.* 13, *no. 1,* pp. 425-448, Jan, 1992. See also http: //netlib2.cs.utk. edu/linalg/lalqmr.

[53] E. Rothberg and A. Gupta, "Parallel ICCG on a hierarchical memory multiprocessor-- addressing the triangular solve bottleneck," *Parallel Comput.,* Vol. 18, pp. 719-741, 1992.

[54] M. J. Grote and Thomas Huckle, "Parallel Preconditioning with Sparse Approximate Inverses, accepted for publication in SIAM *J. of Scientific Computing,* 1996. See also http://www-sccm.stanford.edu/Students/grote.html.

[55] C. H. Walshaw, M. Cross, and M. G, Everett, "A localized algorithm for optimizing unstructured mesh partitions," *international Journal Of Supercomputer Applications And nigh Performance Computing,* Vol 9, No. 4, pp.280-295, 1995.

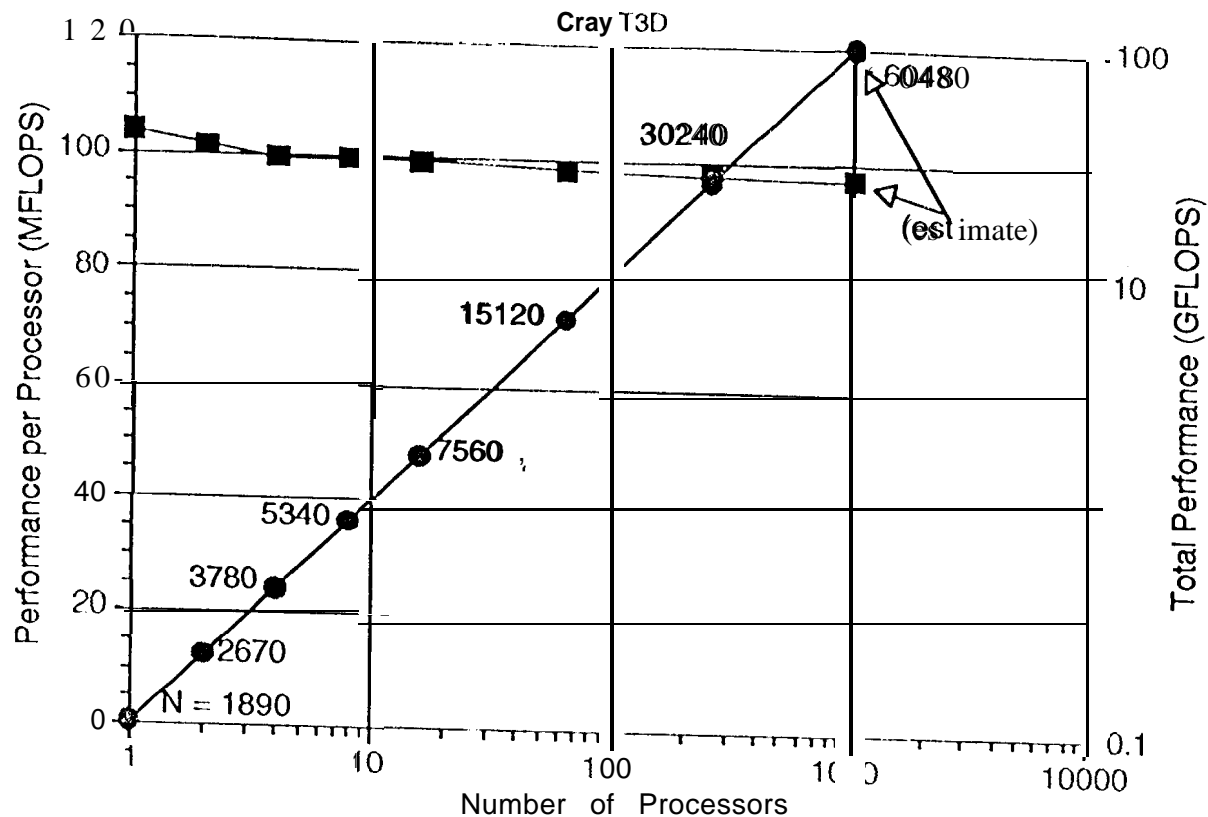Figure 1. Performance for BSOLVE (includes factoring matrix, estimating condition number, and solving for one right hand side.)

TIME FOR LU FACTORIZATION
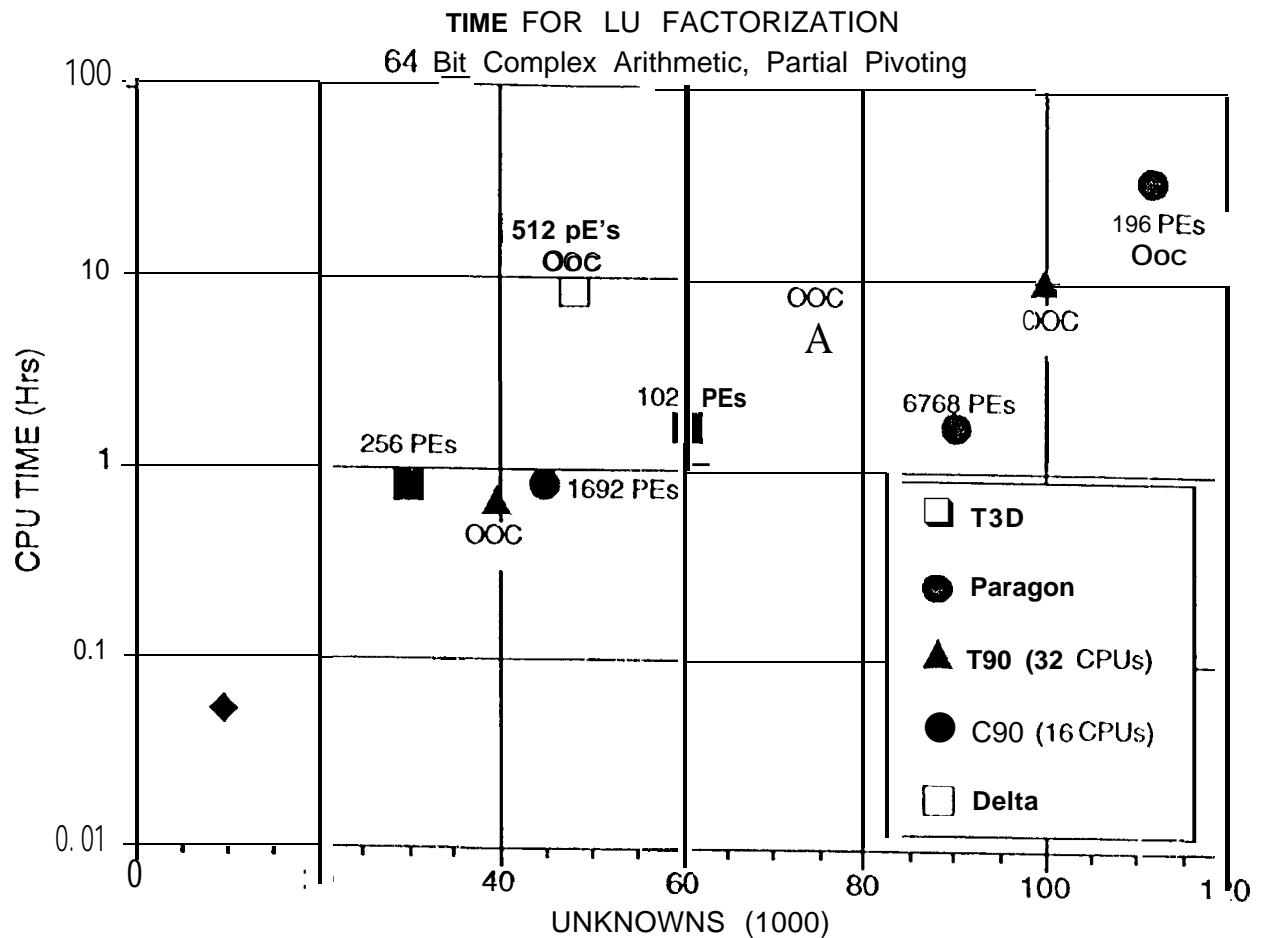64 Bit Complex Arithmetic, Partial Pivoting

Figure 2. CPU time vs. number of unknowns for 64 bit complex dense direct solvers with partial pivoting for various machines (Cray T3D, T90 and C90; Intel Paragon and Delta). processing elements (PEs) used in calculation shown for parallel computers. In-core and out-of-core (OOC) results specified.
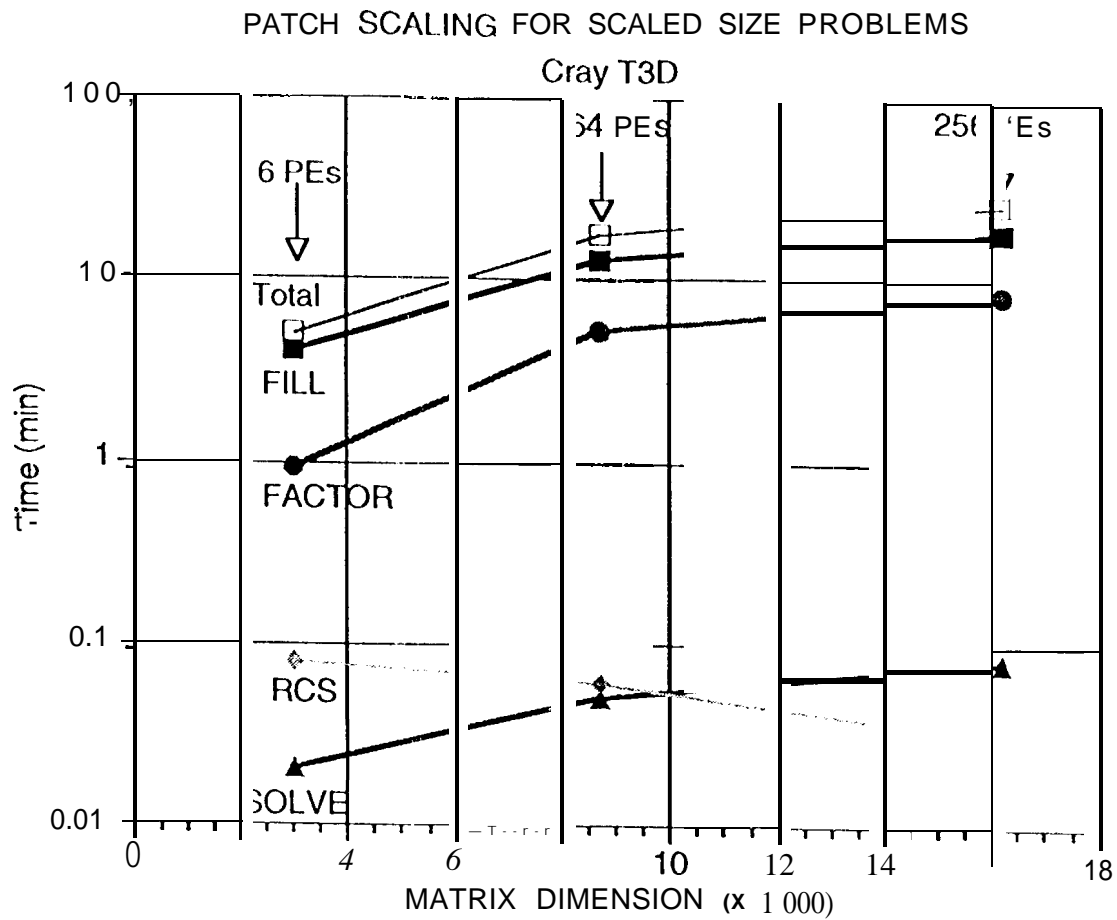
PATCH SCALING FOR SCALED SIZE PROBLEMS

Cray T3D

Figure 3. Scalability for PATCH code for scaled size problems.

PATCH SCALING FOR FIXED SIZE PROBLEMS
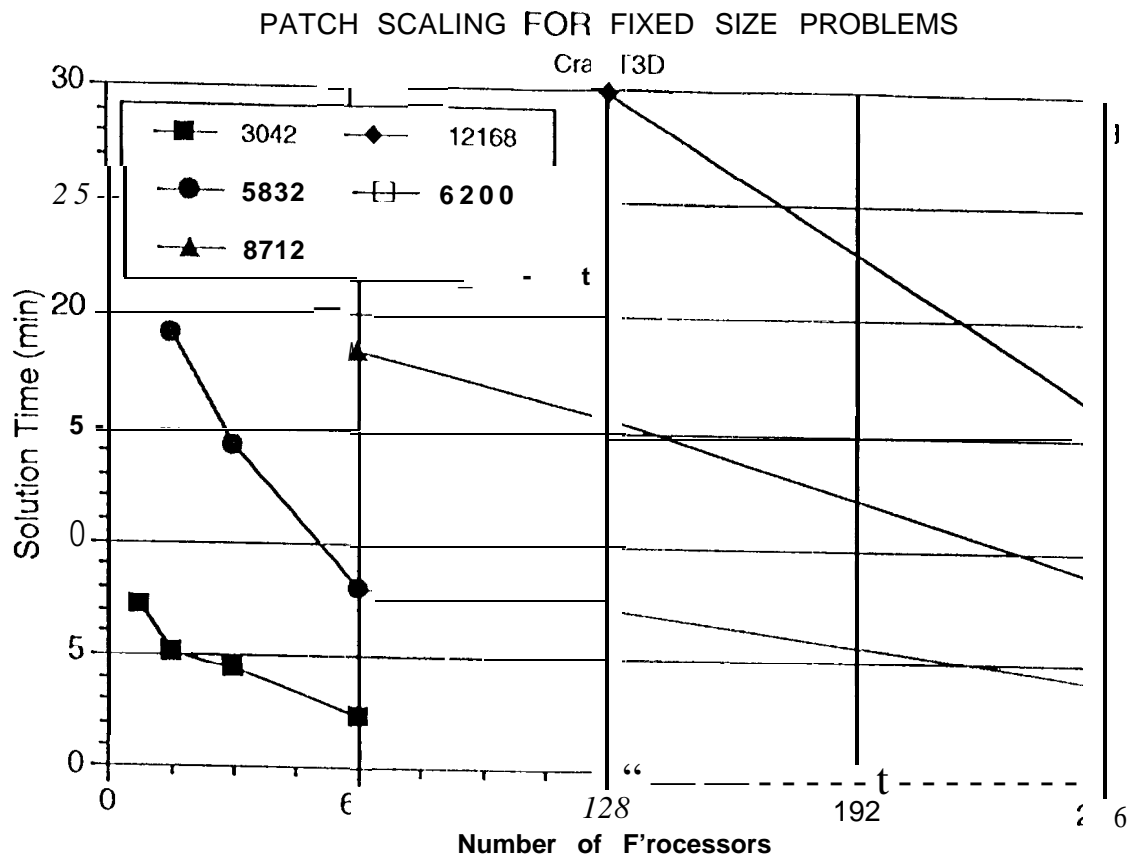
Figure 4. Scalability for PATCH code for fixed size problems. The 16,200 unknown case was only run on 256 processors.

Figure 5. Non-zero matrix structure of typical finite element simulation. Left, original structure: right, structure after reordering to minimize bandwidth.
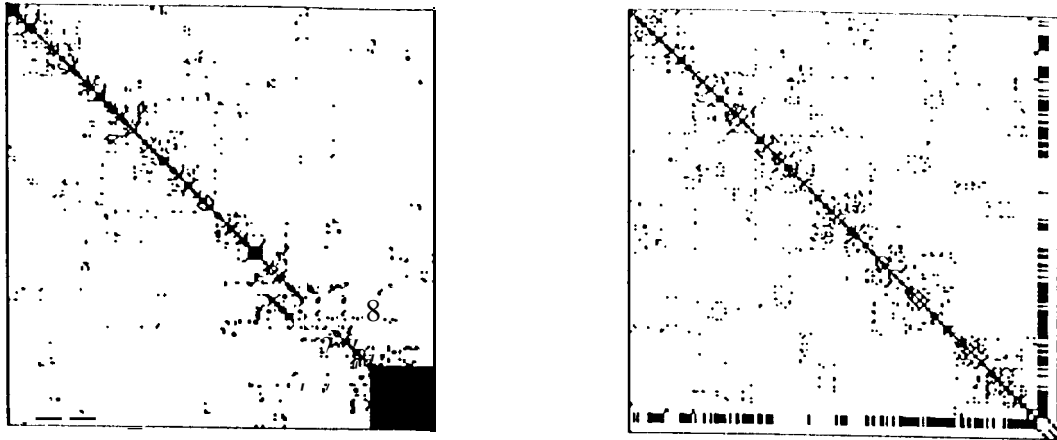
Figure 6. Non-zero matrix sparsity structure for system with dense surface integral equation boundary condition applied (left), and surface of revolution integral equation boundary condition (right). The mesh has 5,343 edges with 936 of those on the boundary.

SCALING OF SPARSE SLICE ALGORITHM
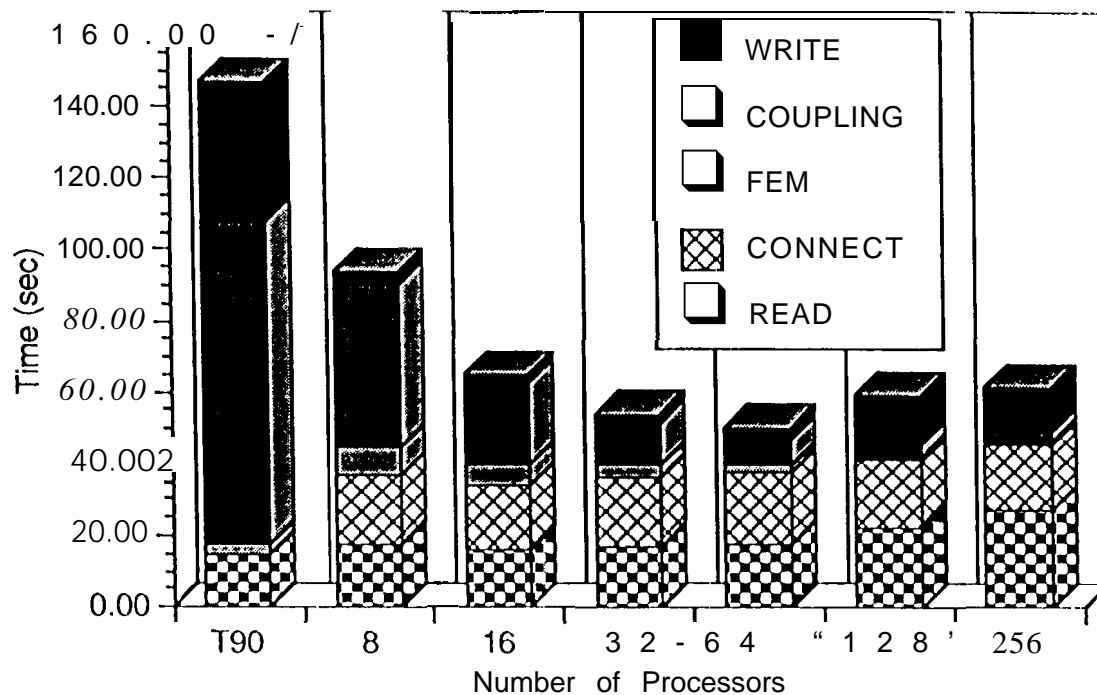
Figure 7. Computation time and scaling for a relatively small simulation (dielectric cylinder with 43,791 edges, radius= 1 cm, height== 10 cm, permittivity = 4.0 at 5.0 GHz). First column shows time for single processor T90. Times on T90 for CONNECT and FEM have been combined.
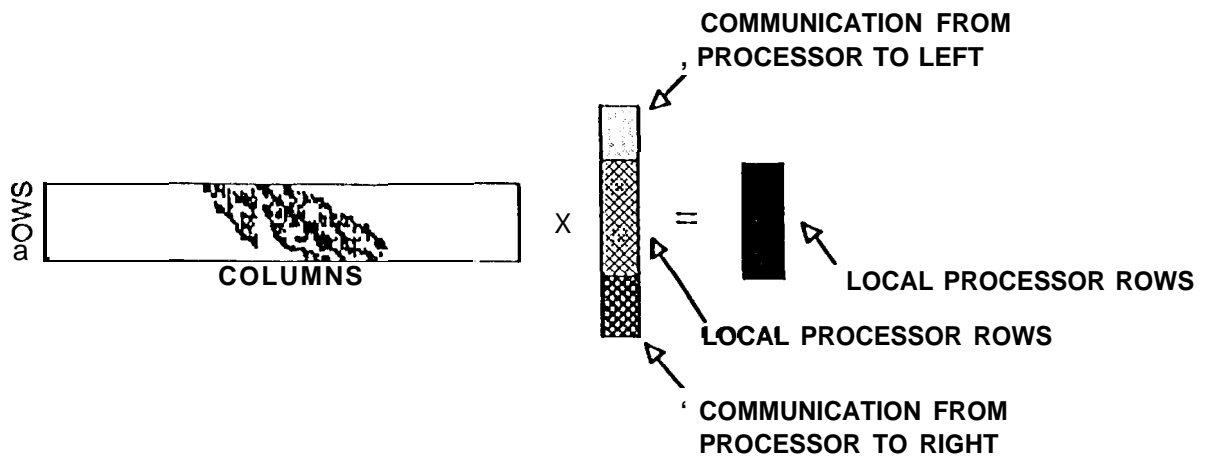
Figure 8. Local sparse matrix-dense vector multiply graphically displayed.

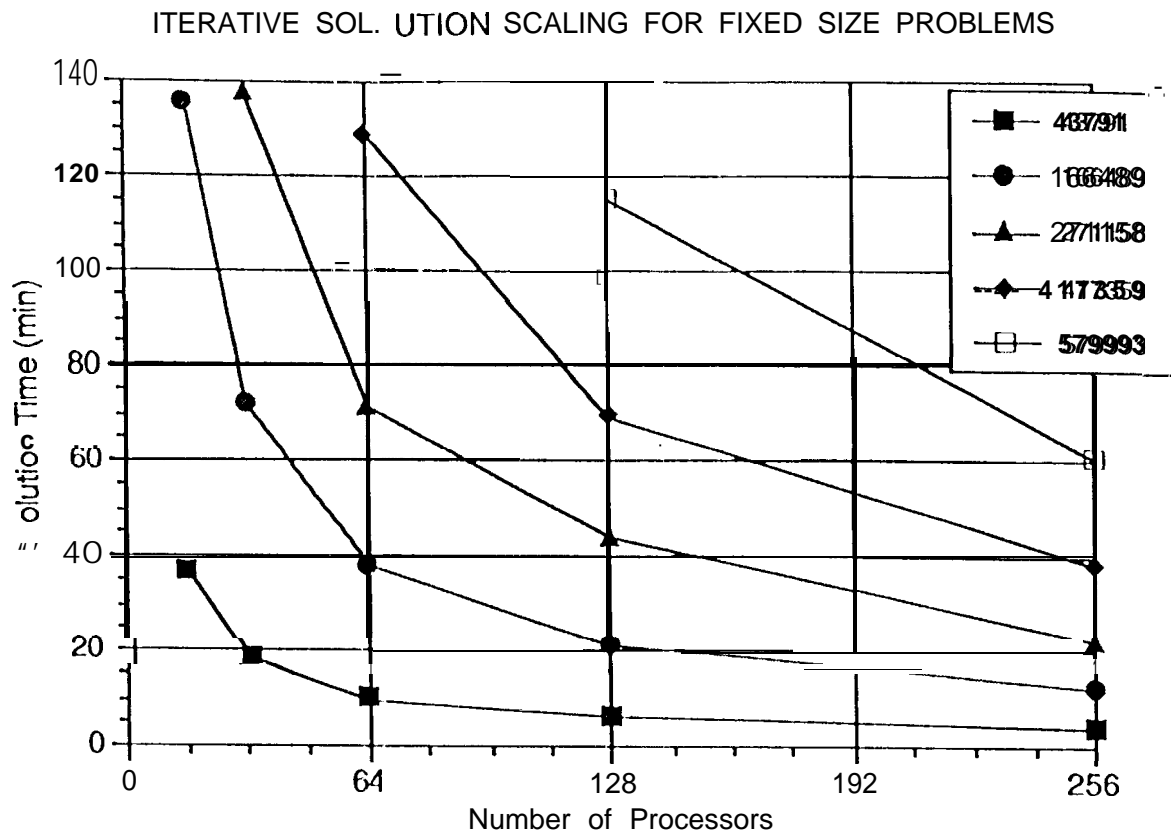ITERATIVE SOL. UTION SCALING FOR FIXED SIZE PROBLEMS

Figure 9. Time of convergence for five different problems. The time shown is the total execution time for the solver on different numbers of processors. The C matrix had 116 columns in each case.
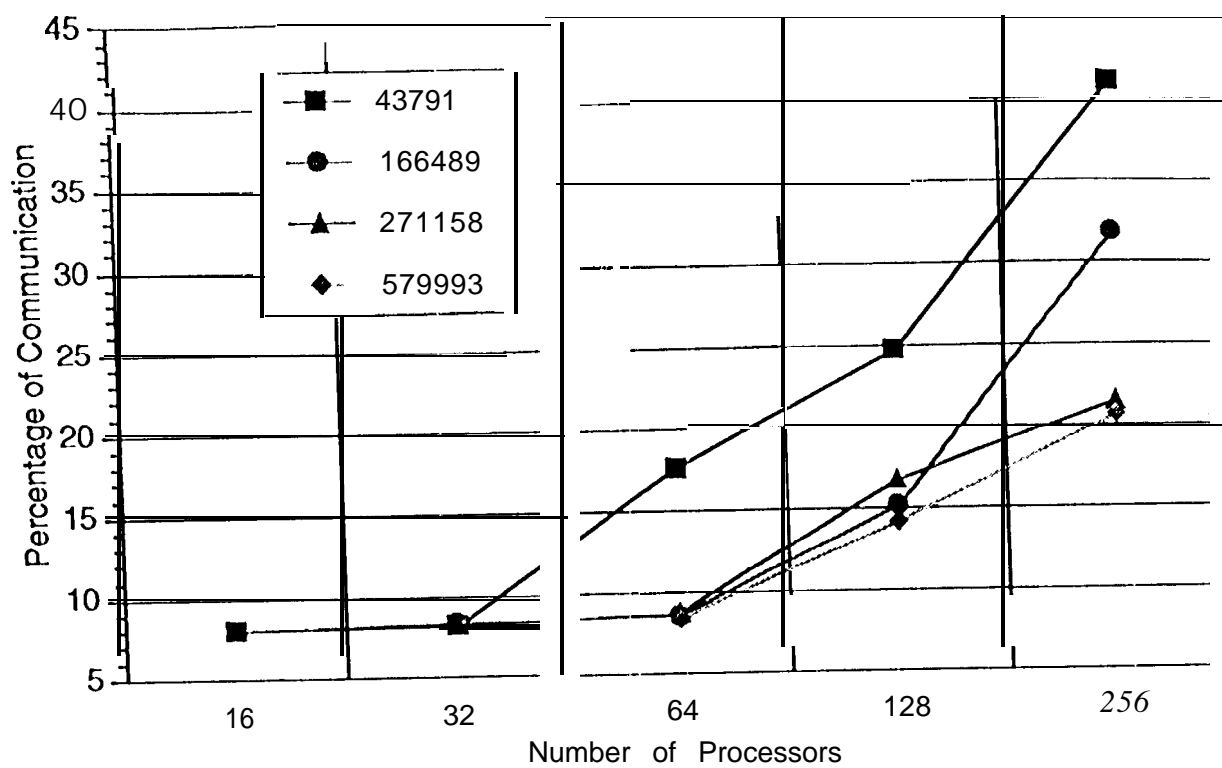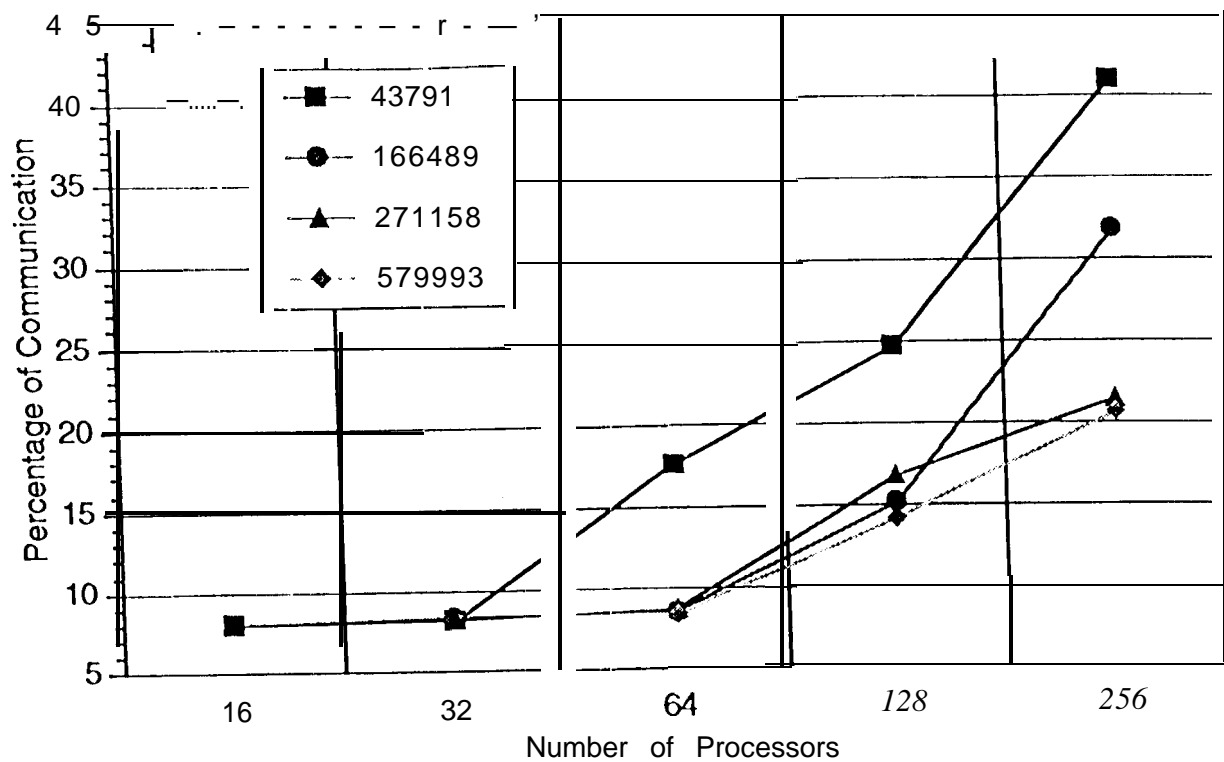
Figure 10. Percentage of communication versus number of processors for parallel matrix vector multiply, for four different size (number of edges) meshes of dielectric cylinder.

Figure 10. Percentage of communication versus number of processors for parallel matrix vector multiply, for four different size (number of edges) meshes of dielectric cylinder.